MASTER EN SCIENCES INFORMATIQUES

Calcul parallèle : les ordinateurs MIMD à mémoire distribuée

Bouvry, Pascal

Award date:
1991

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix,
Institut d'Informatique,
21 rue Grandgagnage,
5000 NAMUR.

Calcul parallèle :

les ordinateurs MIMD à mémoire distribuée.

Pascal BOUVRY

Promoteur : Baudouin LE CHARLIER.

Mémoire présenté en vue
de l'obtention du titre
de Licencié et Maître
en Informatique.

Année Académique 1990-1991.

Abstract

Le calcul parallèle prend de jours en jours une plus grande importance, ce mémoire explore une partie de ce nouveau domaine. Différents types d'architectures coexistent sur le marché. Les ordinateurs parallèles à mémoire distribuée sont caractérisés, entre autres, par le réseau d'interconnexion reliant les différents processeurs. Ceux-ci se synchronisent et communiquent entre eux à l'aide de messages. Les transputers d'Inmos sont des processeurs conçus pour former facilement des réseaux. Un exemple de réseau de ce type est le Méganode formé de 128 transputers de travail et appartenant au laboratoire de recherche LMC-IMAG. L'équipe de chercheurs de ce laboratoire a développé un environnement de programmation pour cette machine comprenant : un langage de description de réseaux, une bibliothèque C permettant de gérer le parallélisme et un debugger couplé à celle-ci. Réaliser des algorithmes parallèles n'est pas simple et il faudra attendre que des outils de parallélisation automatique soient disponibles avant que les architectures distribuées se généralisent.

Nowadays, the researches around parallel computing are growing, this master thesis explores a few of this new domain. Parallel computers with distributed memory are characterised by their interconnection network which binds the processors. These use messages for their synchronisation. The processors, called Transputers, are made by Inmos and have a great ability to form networks. A network of this kind is the Méganode which uses 128 transputers and belongs to the LMC-IMAG research laboratory. The team of this laboratory has developped a programming environment for this machine : a network description language, a C library for the management of parallelism and a debugger coupled with this library. Parallel programming isn't easy and a lot of tools for automatic parallelisation have to come before the massive launch of distributed computers.

Remerciements

Je tiens à remercier tout d'abord le promoteur de mon mémoire, le professeur Baudouin Le Charlier pour la confiance qu'il a bien voulu placer en ma personne tout au long de ce travail et les corrections apportées à ce mémoire.

Ensuite je tiens à remercier les membres du laboratoire LMC-IMAG de Grenoble ainsi que certains appartenant au laboratoire LGI-IMAG, en particulier : le docteur Jean-Louis Roch qui a pris en charge le suivi du développement du debugger pour Méganode, le docteur Gilles Villard, concepteur de Tn_Rita, pour les précieux conseils qu'il a su me prodiguer et les modifications qu'il a bien voulu apporter à Tn_Rita afin de supporter le debugger ; messieurs Philippe Michallon et Patrick Raynal qui m'ont aidé à "debugger" le debugger ; le professeur Brigitte Plateau, le professeur Denis Trystram, monsieur François Vincent, monsieur Patrick Raynal et monsieur Christian Tricot de la société Archipel pour les nombreuses discussions et recherches entreprises ensemble dans le domaine des environnements distribués ; le professeur Yves Robert et le professeur Denis Trystram pour leurs qualités pédagogiques et ce plus particulièrement dans le cadre du cours d'algorithmique parallèle donné en DEA à l'ENSIMAG.

Sommaire

Chapitre I. Introduction 1

Chapitre II. Les ordinateurs parallèles 3

Introduction 4
Taxonomie 5
La vision SIMD 5
Le pipeline 7
Principe 7
RISC versus CISC 9
Les processeurs équipés de pipeline 10
La vision MIMD 10
La vision data-flow 11
Les réseaux systoliques 12
Les ordinateurs MIMD (à instructions et données multiples) 13
Les multiprocesseurs à mémoire partagée 13
Présentation 13
Le partage du bus 14
L'algorithme à priorités statiques 14
L'algorithme à division de temps 14
Les algorithmes à priorités dynamiques 15
L'algorithme FCFS (first-come, first-served) 15
Les ordinateurs parallèles à mémoire distribuée 16
Les réseaux fixes 16
L'hyper-cube 17
L'anneau 19
La grille 20
Les arbres 20
Les réseaux reconfigurables 21
Les réseaux complètement connectés 22
Communication inter-processeurs 23

OTO....24
OTA (diffusion)....25

Chapitre III. La synchronisation, la communication et l'exclusion mutuelle....26

Introduction....27
Les sémaphores....28
Définition....28
Utilisation....29
Implémentation....29
Les sémaphores faibles....29
Les sémaphores forts....30
Critique....30
Exemple....31
Les moniteurs....32
Définition....32
Implémentation....32
Critique....33
Exemple....34
Les messages....35
Définition....35
Nomination de l'émetteur et du récepteur....35
Nomination directe....35
Nomination globale....36
Les ports....36
Distinction synchrone/asynchrone....36
Définition....36
Asynchrone....37
Asynchrone/synchrone....37
Synchrone/asynchrone....38
Synchrone....38
Les commandes gardées....38
Critique....38
Exemple....39

Chapitre IV. Les transputers 41

Introduction. 42
Un langage : OCCAM 42
Présentation 42
Exemple 44
Le transputer T800 44
Présentation 44
Processeur d'entier 46
Unité flottante 46
Mémoire 47
Horloges 47
Liens 47
Le transputer T9000 49

Chapitre V. Un réseau de transputers : le Méganode 50

Le projet ESPRIT 1085 51
Description de l'architecture 52
Configuration et utilisation du réseau en C3L 55
Modèle et extension du C 55
Configuration du réseau 56
Exécution d'un programme 59

Chapitre VI. Un environnement de développement C sur réseaux de transputers.. 60

Introduction 61
Un outil de description de réseau : Tn_Rita 62
Une extension de C : Ouf 64
Les tâches et les processus. 65
Les processus 65
Les canaux 66
L'horloge 67
Les sémaphores 67

Un debugger 67
Origine et présentation 67
Fonctionnalités attendues 68
Problèmes rencontrés 68
Solution proposée et fonctionnalités offertes 69
Implémentation 72
Améliorations 75

Chapitre VII. Programmation d'un réseau de transputer 76

Introduction 77
Qualité d'un algorithme parallèle 78
Contrôle du parallélisme 80
Ferme de processeur 80
Principe 80
Technique d'implémentation 81
Exemple 82
Parallélisme géométrique 82
Principe 82
Technique d'implémentation 83
Exemple 83
Description 83
Programme en pseudo-langage 85
Programme en OUF 86
Complexité de l'algorithme 88
Parallélisme algorithmique 89

Chapitre VIII. Conclusion 91

Bibliographie 92

Index 95

I. Introduction

La miniaturisation et l'accroissement des performances des processeurs se trouvera un jour stoppée par les limites imposées par la physique. Le ralentissement du gain en vitesse des circuits électroniques a d'ailleurs déjà commencé. Comment pourrons-nous alors combler nos besoins en puissance de calcul? L'unique possibilité qui nous sera offerte sera de multiplier le nombre de processeurs. Pourquoi ne pas explorer dès aujourd'hui une partie de ce qui sera le lot des informaticiens de demain? C'est une partie de ce nouveau monde que je vous propose de découvrir au moyen de ce mémoire. Je ne vais pas examiner le cas où l'on essaye d'exploiter au mieux des réseaux d'ordinateurs, mais celui où l'on travaille avec un ensemble de processeurs fortement couplés, c'est-à-dire que le temps pris par une communication entre deux machines n'est pas incomparablement plus long que le temps pris par l'exécution d'une instruction.

L'évolution technologique de la dernière décennie a permis de réduire les coûts des divers composants d'un calculateur en même temps qu'elle augmentait les performances à la fois en temps de calcul, en volume et en fiabilité. Ce qui a permis l'apparition d'architectures multi-processeurs exploitables efficacement.

Dans le chapitre II, nous allons examiner quels sont les grands principes, utilisés au niveau des architectures, qui permettent de mettre en oeuvre le parallélisme. Nous verrons les ordinateurs synchrones constitués d'un ensemble de processeurs travaillant de manière synchrone, la notion de pipeline où des processeurs sont connectés linéairement, les machines MIMD qui comprennent un ensemble de processeurs possédant chacun leur propre programme ; nous citerons les machines data-flow et les réseaux systoliques. Nous développerons le cas des machines MIMD à mémoire distribuée.

Le but du chapitre III est d'étudier les problèmes engendrés par la gestion d'un ensemble de processeurs et trois techniques classiques utilisées pour les résoudre : les sémaphores, les moniteurs et les messages.

Dans le chapitre IV, nous verrons les Transputers. Ces processeurs peuvent être connectés en réseaux et possèdent de base de nombreuses possibilités.

Le chapitre V décrit un réseau de transputers appelé Méganode. Nous allons voir son origine, ses caractéristiques architecturales et les outils de base fournis avec celui-ci.

Pour améliorer le quotidien des programmeurs et remédier aux faiblesses des outils de base fournis avec le méganode, le laboratoire LMC a conçu et réalisé un série d'outils : une file d'attente pour le méganode, un outil de description de réseaux de transputers (Tn_Rita), une interface abstraite (OUF) reprenant les différentes améliorations apportées au C par les C parallèles et le debugger couplé à cette interface. Nous allons étudier ces différents outils dans le chapitre VI.

Le but du chapitre VII est simplement de montrer quelques aspects de la programmation d'un réseau de transputers.

II. Les ordinateurs parallèles

Introduction 4
Taxonomie 5
La vision SIMD 5
Le pipeline 7
Principe 7
RISC versus CISC 9
Les processeurs équipés de pipeline 10
La vision MIMD 10
La vision data-flow 11
Les réseaux systoliques 12
Les ordinateurs MIMD (à instructions et données multiples) 13
Les multiprocesseurs à mémoire partagée 13
Présentation 13
Le partage du bus 14
L'algorithme à priorités statiques 14
L'algorithme à division de temps 14
Les algorithmes à priorité dynamique 15
L'algorithme FCFS (first-come, first-served) 15
Les ordinateurs parallèles à mémoire distribuée 16
Les réseaux fixes 16
L'hyper-cube 17
L'anneau 19
La grille 20
Les arbres 20
Les réseaux reconfigurables 21
Les réseaux complètement connectés 22
Communication inter-processeurs 23
OTO 24
OTA (diffusion) 25

A. Introduction

La dernière décennie a vu apparaître de nombreuses architectures nouvelles, les classifier n'est pas un travail simple car beaucoup de ces nouveaux ordinateurs sont hybrides, c'est-à-dire qu'ils tirent parti, en même temps, de différentes approches. On retrouve dans la littérature des définitions différentes pour les mêmes termes, ce qui ne facilite pas non plus la tâche du classificateur.

Dans ce chapitre, nous allons examiner quels sont les grands principes[1], utilisés au niveau des architectures, qui permettent de mettre en oeuvre un traitement simultané de différentes informations, ce qu'on appelle aussi parallélisme. Nous verrons les ordinateurs synchrones constitués d'un ensemble de processeurs travaillant de manière synchrone, la notion de pipeline où des processeurs sont connectés linéairement, les machines MIMD qui comprennent un ensemble de processeurs possédant chacun leur propre programme, la machines data-flow qui exécutent les instructions suivant la disponibilité des opérandes et les réseaux systoliques qui sont constitués d'un ensemble de petits automates. Nous développerons le cas des machines MIMD à mémoire distribuée.

Notons qu'il existe aussi du pseudo-parallélisme qui est un mécanisme basé sur le partage du temps de calcul d'un processeur entre plusieurs programmes, ce qui permet de donner à l'utilisateur l'illusion du parallélisme.

[1] Références bibliographiques : [Banatre 90], [Bell 89], [Dewar 90], [Duncan 90], [Hwang 84], [Perrot 87].

B. Taxonomie

1. La vision SIMD

Les multi-processeurs basés sur une architecture SIMD (*Single Instruction Multiple Data*, cf. figure 2.1) sont des ordinateurs possédant plusieurs composantes processantes (unités arithmétiques et logiques) et une seule unité de contrôle. Ces processeurs sont synchronisés afin d'effectuer la même opération au même moment. Chaque processeur possède ses propres données. Notons qu'il existe une variante appelée SPMD (*Single Procedure Multiple Data*) où les processeurs partagent un même programme.

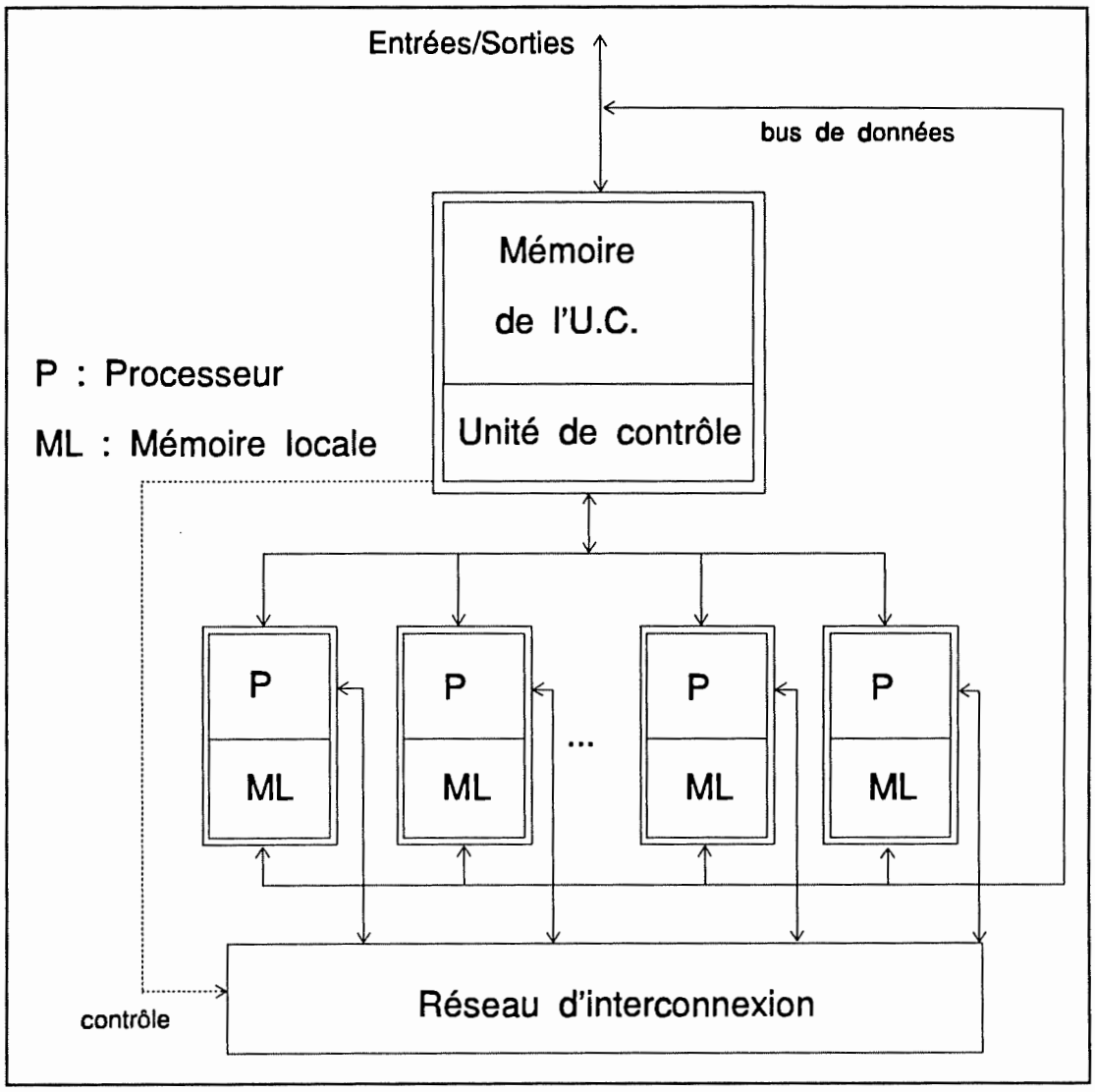


Figure 2.1. Exemple d'architecture SIMD.

L'architecture SIMD illustrée par la figure 2.1 est celle que l'on retrouve dans l'ordinateur Illiac-IV. Cette configuration comprend N composantes processantes qui sont toutes sous le contrôle d'une UC (unité de contrôle). Une composante processante est une unité arithmétique et logique qui possède des registres de travail et une mémoire locale. Le système et les programmes utilisateurs sont exécutés sous le contrôle de l'UC. Les programmes utilisateurs sont chargés dans la mémoire de l'UC à l'aide d'une source externe. Le but de l'UC est de décoder toutes les instructions et de déterminer où les instructions doivent être exécutées. Les instructions scalaires ou de contrôle sont exécutées directement par l'UC. Les instructions vectorielles (qui concernent des vecteurs de données) sont diffusées vers les composantes processantes qui les exécuteront en parallèle.

Toutes les composantes processantes exécutent la même fonction de manière synchrone sous le contrôle de l'UC. Les opérandes sont distribuées aux mémoires des processeurs avant l'exécution parallèle. Les données distribuées peuvent être chargées dans les mémoires grâce à une source extérieure ou via l'UC qui les diffuse en utilisant le bus de contrôle. Chaque composante processante peut être activée ou désactivée durant un cycle d'instruction. Un vecteur de masquage est utilisé pour contrôler le statut de chaque composante processante lors de l'exécution d'un instruction vectorielle. Seules les composantes activées exécutent l'opération. Les échanges de données entre composantes processantes sont faites via un réseau de communication inter-processeurs qui s'occupe des fonctions de routage des données.

Un ordinateur SIMD est normalement interfacé à un processeur hôte via l'UC. Les fonctions de cet ordinateur comprennent la gestion des ressources, des périphériques et la supervision des entrées/sorties.

Voyons, l'exemple de la boucle Fortran suivante qui peut être facilement traitée par un SIMD possédant 64 composantes processantes :

```
DO I=1, 64
     C(I) = A(I) + B(I)
CONTINUE
```

Il suffit que le contrôleur demande à chaque processeur i ($1 \leq i \leq 64$) d'exécuter une addition avec comme opérandes A(i) et B(i) qui se trouvent dans sa mémoire locale.

2. Le pipeline

a) Principe

Dans le cas du pipeline, on connecte linéairement les processeurs. A chaque processeur on fournit un code exécutable et on fait passer un flux de données à travers la chaîne des processeurs. Les différentes parties du pipeline sont appelées étages.

Les données passent dans le premier processeur et les résultats sont emmagasinés dans un bloc mémoire accessible par le second processeur. Le second processeur stocke ses résultats dans un bloc mémoire accessible par le troisième, et ainsi de suite.

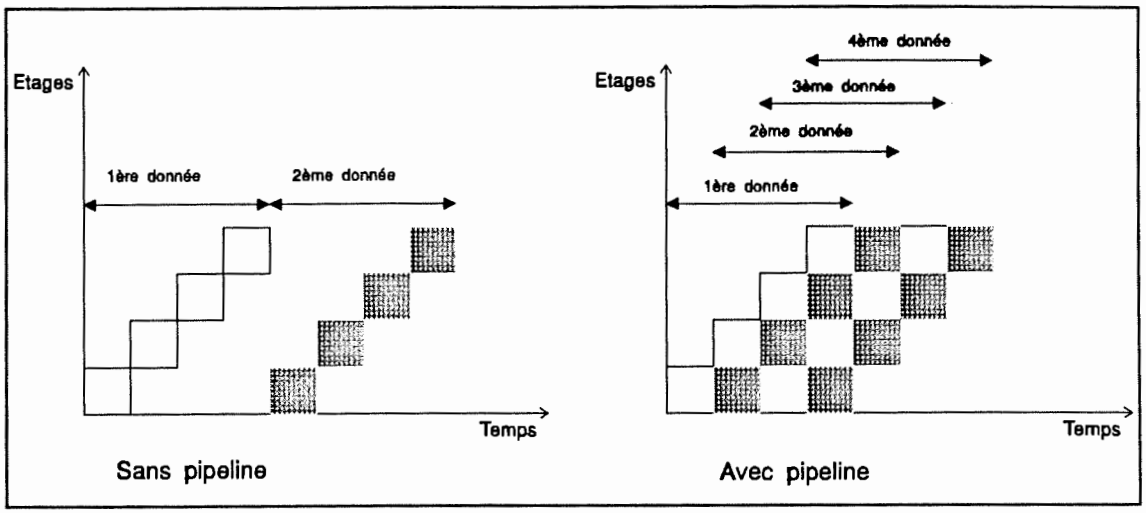


Figure 2.2. Fonctionnement d'un pipeline.

Admettons que nous possédions P processeurs et que chacun de ceux-ci calcule un résultat à partir d'opérandes. Admettons que chaque processeur travaille à la même vitesse, c'est-à-dire qu'ils mettent T micro-secondes pour un calcul, et qu'il ne peut y avoir qu'un seul processeur qui travaille à un moment précis, alors le temps total pris pour s'occuper de N séries d'opérandes sera de NxPxT micro-secondes, ce qui correspond à 1000/(NxPxT) opérations par seconde.

Si maintenant, on permet aux processeurs de travailler simultanément, le temps pris par les calculs sera de PxT+(N-1)xT. Le premier terme représentant le temps pris pour le traitement de la première série. Le nombre d'opérations par seconde est donc de 1000/((P+N-1)xT), ce qui représente une amélioration par rapport au premier cas de (NxP)/(P+N-1).

Ces résultats théoriques ne sont pas toujours réalistes. En effet, rares sont les cas où l'on retrouve toute une série d'opérations découpables en une série d'étapes identiques. Si lors d'un calcul, on a besoin des résultats d'un autre calcul qui est encore dans le pipeline, on peut se voir obligé de vider celui-ci, ce qui est une perte de temps.

b) RISC versus CISC

Suite aux difficultés de programmation rencontrées avec les premiers ordinateurs, les scientifiques ont conçu toute une série de langages de hauts niveaux. Par après, les concepteurs de micro-processeurs ont développé des circuits reflétant le plus possible les possibilités de ces nouveaux langages. Ils voulaient faciliter l'écriture de compilateurs et pensaient améliorer le temps d'exécution des programmes par le simple fait que les instructions cablées (réalisées au niveau matériel) sont plus performantes que les instructions logicielles. Ces diverses cogitations ont donné naissance aux processeurs à jeux d'instructions complexes (CISC, i.e., 80386, 68030, ...).

Un processeur peut être décomposé en niveaux logiques. Si le fait de rajouter un ensemble d'instructions amène la construction d'un nouveau niveau, le gain en performance obtenu par l'utilisation de ces instructions doit au minimum égaler la perte subie par l'ajout du niveau (le décodage de chaque instruction passant par ce niveau). Or l'expérience nous montre que peu de compilateurs utilisent l'ensemble des instructions. SUN déclare que sur ses machines à base de 68020, le compilateur C d'origine n'utilise que 30% des instructions. De plus même si certains compilateurs utilisaient ces instructions, on n'imagine pas les retrouver fréquemment. Ces différentes constatations ont amené une génération de processeurs à jeux d'intructions réduit (RISC, i.e., T800, Sparc, ...). Notons qu'il ne s'agit pas simplement d'un ensemble réduit d'instructions mais surtout d'instructions simples. Les modifications par rapport à un processeur classique se retrouvent principalement dans la limitation des possibilités d'adressage et dans la favorisation des instructions manipulant des registres.

De nombreux processeurs RISC sont équipés de pipeline car l'exécution d'une instruction RISC peut être plus facilement découpée en phases ayant des durées analogues, ce qui n'était pas le cas des processeurs CISC.

c) Les processeurs équipés de pipeline

La technique du pipeline peut être utilisée au sein d'un processeur RISC. L'analyse et l'exécution d'une instruction peuvent être découpées en plusieurs étapes, par exemple [DEWAR 90] :

IF (*Instruction Fetch*) : le chargement d'une instruction dans la mémoire du processeur.

ID (*Instruction Decode*) : le décodage de l'instruction.

OF (*Operand Fetch*) : Les opérandes (souvent certains registres) sont stockées dans un emplacement approprié du processeur où les opérations sont effectuées.

OP (*Operate*) : L'opération est exécutée durant cette phase.

WB (*Write Back*) : Les résultats de l'opération sont réécrits dans un registre (où à un emplacement mémoire).

Le processeur peut donc effectuer plusieurs opérations à la fois en pipeline. Si on a une instruction qui modifie le pointeur d'instruction, on remarque que le processeur peut exécuter des instructions qui ne devraient pas l'être, en effet, il ne peut pas anticiper la modification du pointeur d'instruction. Ce problème est résolu généralement en réinitialisant le pipeline, ce qui implique une perte de temps.

3. La vision MIMD

Dans le cas des ordinateurs MIMD (*Multiple Instruction Multiple Data*), chaque processeur possède sa propre horloge et son propre code qui s'applique à différentes données.

La majeure partie de ce travail a comme objet principal les machines MIMD à mémoire distribuée, d'une part du fait des opportunités qui m'ont été offertes et d'autre part du fait des possibilités de ce type d'ordinateur. En effet, les nouveaux paradigmes issus de la réflexion des scientifiques concernant ces machines me semblent plus riches et plus

universels que ceux restreints par la rigidité d'une architecture pipeline ou que ceux issus du développement des machines SIMD. Avec un ordinateur de type MIMD rien ne nous empêche de considérer que certains processeurs sont connectés linéairement et et de les utiliser en pipeline ; de même, nous pouvons aussi fournir un code différent à plusieurs processeurs et simuler un ordinateur SIMD. La flexibilité et la capacité de simulation des MIMDs représente un attrait considérable.

Nous étudierons les ordinateurs MIMD plus en profondeur dans la section et les chapitres suivants.

4. La vision data-flow

Les architectures *data-flow* cherchent à exploiter au maximum le parallélisme d'un algorithme en s'affranchissant des contraintes liées à l'ordre d'exécution des instructions. Le principe de base est d'autoriser l'exécution d'une instruction dès que ses opérandes sont disponibles. Le début d'une instruction ne dépend que de la disponibilité des données et non pas de sa position dans le programme : l'exécution est liée aux contraintes de dépendances entre données.

Un programme pour de telles architectures est représenté par le graphe du flot de données. La figure suivante montre le graphe correspondant à l'expression r=(a+b)*(c+d). Une instruction est donc une partie du graphe constituée d'un noeud (l'opérateur), des arcs entrants (les opérandes) et des arcs sortants (les résultats). Une instruction est active dès que les opérandes ont une valeur.

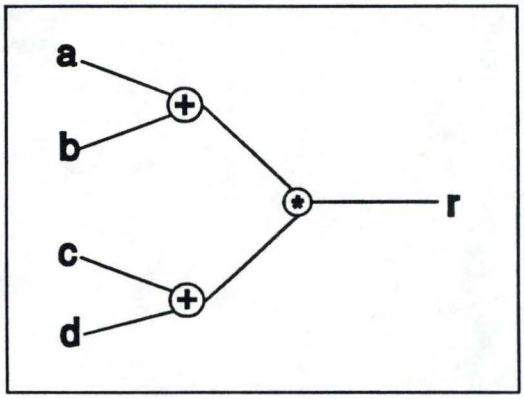


Figure 2.3. Traitement effectué par un data-flow.

L'organisation d'un ordinateur data-flow comporte :

- une unité de mémoire contenant les opérandes et les instructions non-actives ;
- une unité de recherche des instructions actives ;
- une file des instructions actives en attente d'exécution ;
- une unité de traitement avec plusieurs processeurs ;
- une unité de mise à jour dont le rôle est d'affecter leurs valeurs aux opérandes à l'issue du traitement.

La recherche dans le domaine des data-flow a été très forte dans les années 70, cependant le manque de résultats (d'architectures performantes), fait qu'actuellement cette piste est de moins en moins explorée.

5. Les réseaux systoliques

Le modèle des réseaux systoliques a été introduit en 1978 par Kung et Leiserson pour la conception d'architectures spécialisées.

Les réseaux systoliques sont constitués d'un ensemble de cellules élémentaires, semblables à de petits automates, possédant chacun un code exécutable et connectés de manière à former un graphe local et régulier. L'ensemble du réseau fonctionne de manière synchrone : à l'instant t, les cellules reçoivent des données de leurs voisines en entrée, effectuent les transformations nécessaires sur ces variables et les renvoient, à l'instant t+1, à leurs cellules voisines en sortie.

Le terme réseau systolique provient d'une analogie entre la circulation régulière des données sur le réseau et celle du sang dans un organisme vivant.

C. Les ordinateurs MIMD (à instructions et données multiples)

Les ordinateurs MIMD peuvent être conçus de différentes manières : soit comme un ensemble de processeurs utilisant une mémoire partagée via un bus commun, soit comme un ensemble de processeurs possédant chacun une mémoire locale et reliés par un réseau d'interconnexion. Il existe aussi des solutions tirant parti des deux optiques.

1. Les multiprocesseurs à mémoire partagée

a) Présentation.

Dans un multi-processeur à mémoire partagée, les processeurs peuvent communiquer et coopérer pour résoudre un problème donné. Les communications inter-processeurs peuvent avoir lieu grâce au partage d'une zone de mémoire commune.

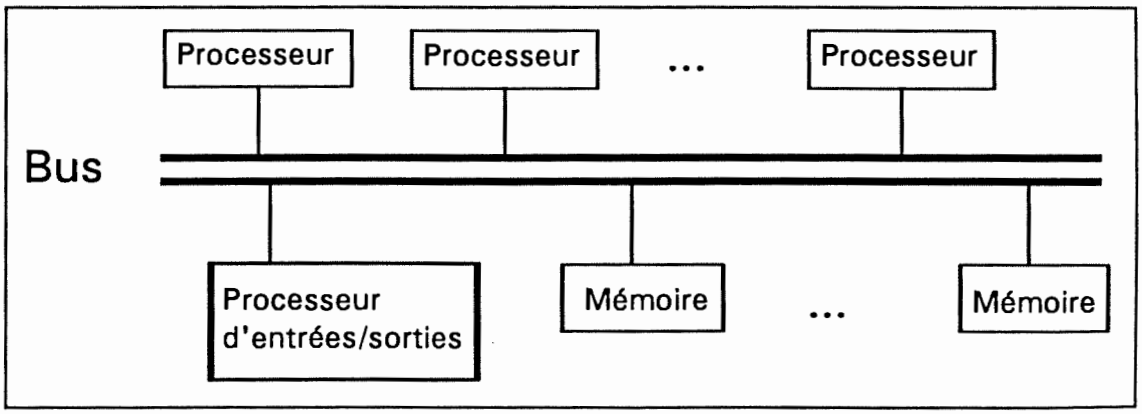


Figure 2.4. Architecture de base d'un MIMD à mémoire partagée.

Les machines parallèles à mémoire partagée (cf. figure 2.4) ont comme principal inconvénient d'utiliser un bus commun aux différents processeurs. Comme ce bus possède une bande passante limitée, le nombre d'accès à ce bus, par unité de temps, est aussi limité, ce qui nous empêche d'augmenter indéfiniment le nombre de processeurs. La technologie actuelle au niveau des bus ne permet d'imaginer que des multiprocesseurs possédant au plus quelques dizaines de processeurs.

b) Le partage du bus.

Comme le bus est un ressource partagée, un mécanisme de résolution des problèmes de contention doit être utilisé. Les méthodes de résolution de conflits peuvent se baser sur des mécanismes de priorité, sur du partage de temps ou sur des files attente FIFO (*first in first out*).

La technologie actuelle (vitesse de traitement) requiert des algorithmes relativement simples pour l'arbitrage du bus. Ces algorithmes sont habituellement implémentés au niveau *hardware*.

(1) L'algorithme à priorités statiques

De nos jours, de nombreux bus digitaux assignent des priorités statiques aux éléments émettant des requêtes. Quand plusieurs éléments, par exemple des processeurs, émettent en même temps une requête, celui avec la priorité la plus haute verra sa demande satisfaite. Cette approche est généralement implémentée grâce à une technique appelée *daisy chaining*, dans laquelle tous les services ont effectivement des priorités statiques suivant leur localisation le long d'une ligne de garantie de contrôle du bus. L'élément le plus proche du contrôleur du bus a la priorité la plus élevée.

(2) L'algorithme à division de temps

Dans cet algorithme la bande passante du bus est divisée en intervalles fixes de temps. Si un élément sélectionné n'utilise pas le laps de temps qui lui est alloué, aucun autre élément ne peut l'utiliser. Cette technique est appelée FTS (*fixed time slicing*) ou TDM (*time division multiplexing*) . Les services fournis à chaque élément avec l'algorithme TDM

sont indépendants de la position de l'élément ou de son identité ; les schémas utilisant cette caractéristique sont appelés symétriques.

Quand le bus n'est pas surchargé, TDM amène des temps de réponse moins intéressants que l'algorithme à priorités statiques, mais possède l'avantage que la disponibilité du service fourni ne dépend pas de cette charge.

(3) Les algorithmes à priorités dynamiques

Les algorithmes à priorités dynamiques permettent de répartir l'utilisation du bus comme dans le cas d'un algorithme TDM mais sans le défaut des forts temps d'attente. Chaque élément possède un niveau de priorité unique et est en compétition pour l'accès au bus, mais les priorités sont changées dynamiquement pour donner une chance à chaque élément d'accéder au bus. Si l'algorithme utilisé pour gérer les priorités ne favorise aucun élément, alors le système répartit équitablement l'accès au bus. De plus, l'utilisation de priorités permet de ne pas avoir l'inefficacité amenée par un découpage fixe du temps.

L'algorithme LRU (*least recently used*) est un algorithme utilisé pour la gestion dynamique des priorités, il fournit la priorité la plus élevée à l'élément qui n'a pas utilisé le bus depuis le plus grand laps de temps. Ceci s'effectue après chaque cycle de bus.

(4) L'algorithme FCFS (first-come, first-served)

Dans l'algorithme FCFS, les requêtes sont simplement traitées dans l'ordre de réception. Ce mécanisme est symétrique car il ne favorise aucun des éléments qui sont connectés au bus. En terme de mesure de performances (répartition équitable et utilisation efficace de la bande passante du bus), FCFS est l'algorithme le plus performant.

Malheureusement, FCFS est difficile à implémenter pour au moins deux raisons. Toute implémentation de FCFS doit posséder un mécanisme d'enregistrement de l'ordre d'arrivée de toutes les demandes qui sont en attente, ce qui n'était pas le cas des algorithmes précédents. De plus, on peut toujours envisager le cas où deux demandes arrivent dans un intervalle de temps si petit qu'il est impossible d'établir l'ordre de leurs arrivées. Vu cela, les différentes implémentations possibles ne peuvent qu'approximer le comportement FCFS. Malgré les difficultés d'implémentation, il est important de mesurer les performances de FCFS pour s'en servir comme indicateur de la meilleure performance possible qu'un algorithme de gestion d'un bus puisse atteindre (suivant les critères de répartition équitable et d'utilisation efficace de la bande passante du bus) .

2. Les ordinateurs parallèles à mémoire distribuée

Dans un ordinateur parallèle à mémoire distribuée, chaque processeur possède un nombre limité de liens (limite physique) le reliant à l'extérieur. Nous pouvons envisager de connecter les liens de manière statique ou de manière dynamique, dans ce cas le réseau est reconfigurable soit lors du chargement soit lors de l'exécution d'un programme.

Il est aussi possible d'imaginer un ordinateur composé d'un ensemble de processeurs possédant chacun une mémoire locale et reliés par un bus. Cette hypothèse nous ramènerait aux limites des machines à mémoire partagée du fait de la bande passante limitée des bus actuels, mais posséderait l'avantage de constituer un réseau complètement connecté de processeurs.

a) Les réseaux fixes

Les connexions statiques[2] paraissent à première vue contraignantes pour l'utilisateur. Cette constatation a amené bien des équipes de recherches à essayer de trouver la topologie la moins contraignante. Cette topologie doit avoir comme principale caractéristique de minimiser le diamètre (le plus long des chemins de taille minimale séparant deux processeurs quelconques du réseau) du graphe des processeurs et devrait être assez générale pour servir de base à d'autres topologies.

Une des facilités offertes par les réseaux fixes est leur aptitude à être facilement analysés mathématiquement.

---

[2] Référence : cours de DEA, "algorithmique parallèle", d'Yves Robert et de Denis Trystram, ENSIMAG 1990.

(1) L'hyper-cube

Un hyper-cube de degré m est un réseau à 2^m sommets, où chaque sommet possède exactement m voisins.

Cette figure est récursive, on peut construire un m-cube à partir de deux (m-1) cubes dont les sommets identiques sont reliés.

Il existe une numérotation commode des sommets, à partir des codes de Gray, où chaque noeud du réseau est un entier codé sur m bits en binaire, dont on déduit tous les voisins en complémentant successivement chaque bit de cette numérotation. Par exemple, le noeud 1 (001) est relié aux noeuds 0, 3 et 5 (respectivement 000, 011, 101).

Un m-cube possède un diamètre intéressant de taille m et il est assez simple d'y plonger, entre autres, un arbre, une grille ou un anneau.

Cependant la dimension de l'hyper-cube est limitée au nombre de liens de processeurs. Nous pouvons remédier à cela en utilisant, par exemple, une topologie appelée CCC (Cube Connecté Cycle) qui consiste à placer à chaque noeud de l'hyper-cube un anneau de processeurs (chaque processeur possédant trois liens : deux pour l'anneau et un pour la connexion à la dimension suivante de l'hyper-cube).

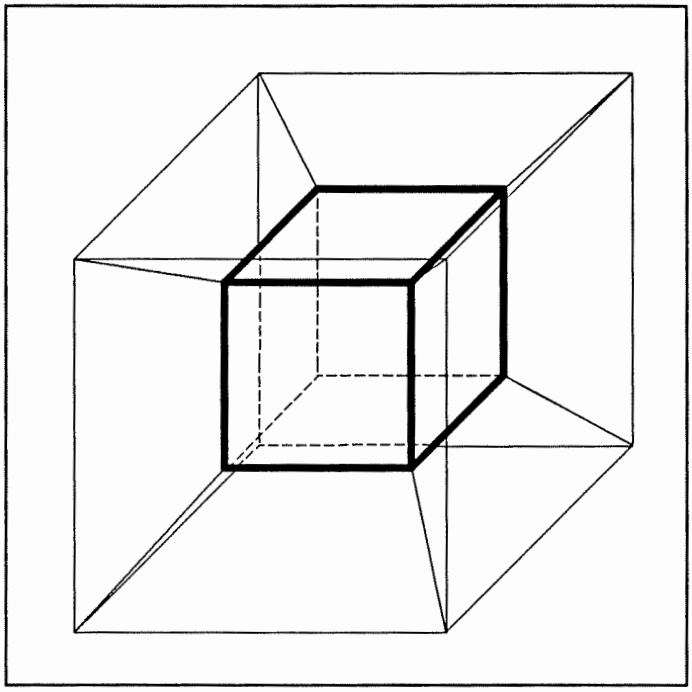


Figure 2.5. Hyper-cube de degré 4

(chaque sommet représentant un processeur).

L'analyse des procédures de communication sur ces réseaux est basée sur les arbres de recouvrement (dont les principaux sont représentés sur la figure suivante). Il s'agit d'un arbre contenant tous les noeuds du réseau. Un arbre binomial sur un m-cube est construit récursivement en reliant les deux arbres binomiaux recouvrant des (m-1)-cubes.

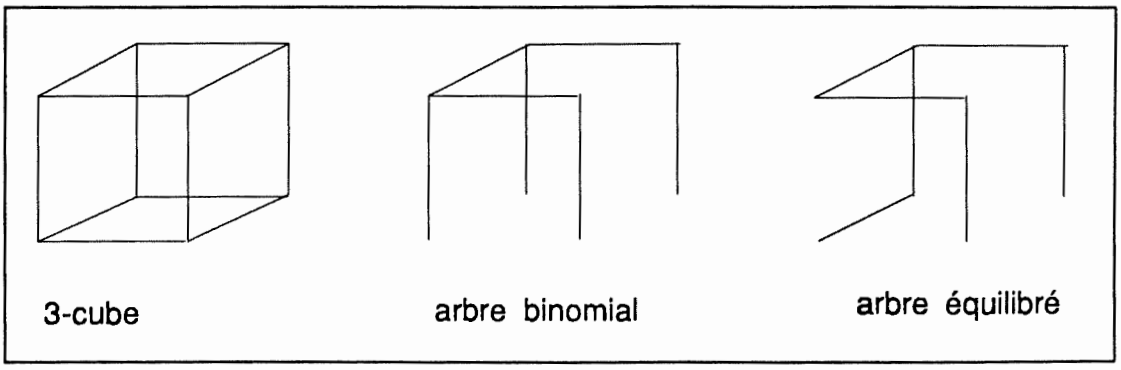


Figure 2.6. Hyper-cube de degré 3 et arbres de recouvrement.

(2) L'anneau

Les topologies en anneau ont été beaucoup utilisées au début, du moins dans un cadre théorique, pour leur simplicité relative quant à la possibilité de donner des résultats de complexité. On obtient facilement un anneau à partir d'une topologie hyper-cube.

Dans un hyper-cube numéroté à l'aide des codes de Gray, chaque position de bit correspond à une direction.

L'idée est de partir d'un sommet quelconque (le noeud 0 sur la figure est choisi pour une plus grande simplicité), et de déterminer les successeurs sur l'anneau en complémentant le bit dont la position est déterminée par les directions correspondant au code de Gray réfléchi.

Soit d(m), la suite des 2^n-1 directions successives définies récursivement par la formule :

d(1)=0

d(m)=(d(m-1) m-1 d(m-1))

Plus concrètement, on prend dans un 3-cube comme directions successives ((010)**2**(010), et ((0102010)**3**(0102010)) pour un 4-cube, etc.

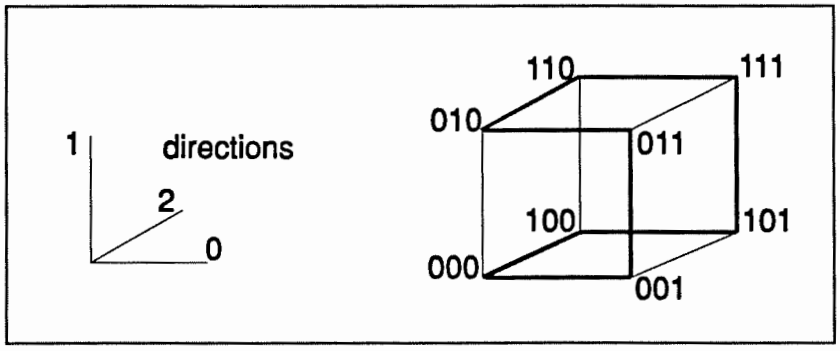


Figure 2.7. Plongement d'un anneau dans un 3-cube.

(3) La grille

D'une manière plus générale, une grille de dimension n (ou n-grille), est une topologie à $\prod_{i=1}^{n} 2^{m_i}$ noeuds, comportant 2^{m_i} processeurs sur chaque direction i (pour i allant de 1 à n) où chaque processeur a 2n voisins. En pratique, on utilise des tores (grille refermée).

L'anneau défini précédemment est une grille de dimension 1 (c'est-à-dire un réseau où chaque noeud a exactement 2 voisins). De même, une grille de dimension 2, de taille nxm, comprend n lignes de m processeurs, chacun ayant 4 voisins (m et n étant en général des puissances de 2). En pratique, une grille est constituée d'anneaux. La grille semble être un relativement bon compromis de par la simplicité des algorithmes (en particulier pour tout ce qui manipule des tableaux).

(4) Les arbres

Les arbres sont souvent employés, surtout dans les stratégies "centralisées" du style "maître-esclaves" ou autres structures hiérarchiques.

b) Les réseaux reconfigurables

Si nous voulons permettre la reconfiguration dynamique, nous pouvons utiliser des circuits permettant de connecter, point à point, n entrées à n sorties, appelés *crossbar switches* (cf. figure 2.8). La configuration (le choix de relier telle entrée à telle sortie) de tels circuits s'effectue, simultanément pour toutes les entrées et sorties, via un processeur de contrôle externe.

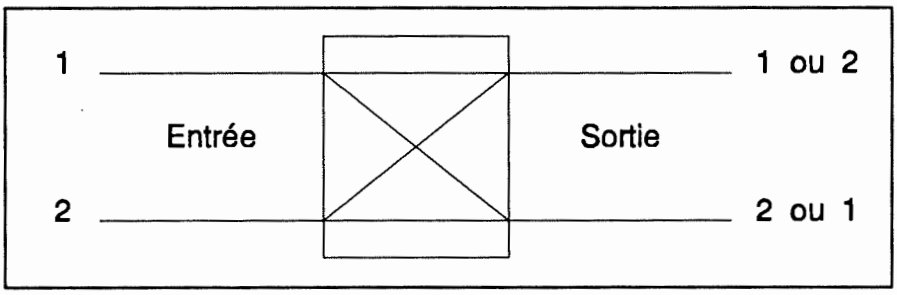


Figure 2.8. Un *crossbar switch* 2x2.

Mais nous nous heurtons à une limite physique : en effet, les *crossbar switches* actuels ne possèdent qu'un nombre très limité d'entrées et de sorties. La solution trouvée pour remédier à ce problème est d'utiliser une structure hiérarchique ; cependant la distance que les signaux doivent parcourir quand ils traversent plusieurs niveaux de la hiérarchie est telle qu'il faut utiliser des répéteurs afin de conserver l'ensemble de l'information.

Les répéteurs sont situés dans des *crossbar switches* spéciaux dont la traversée est coûteuse et on appelle l'action des répéteurs, rééquilibrage des signaux.

La reconfiguration dynamique en cours d'exécution du programme est actuellement extrêmement peu répandue ; ce qui explique que les études effectuées dans le cadre de réseaux à topologies fixes sont très utiles puisque, par exemple, suite à ces études, l'utilisateur possédant un hyper-cube qui aurait envie d'avoir à sa disposition une grille et un anneau sait qu'il est possible de plonger un anneau ou une grille dans un hyper-cube.

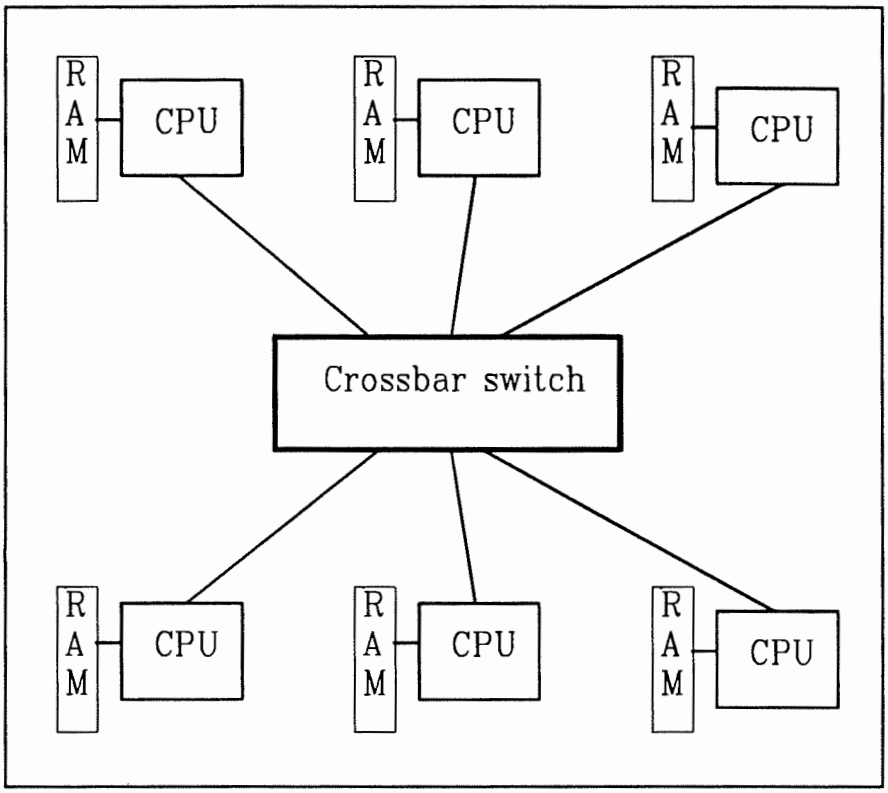


Figure 2.9. MIMD à mémoire distribuée utilisant un *crossbar switch.*

c) Les réseaux complètement connectés

Il est actuellement possible de réaliser des réseaux complètement connectés comportant un nombre restreint de processeurs.

On peut, par exemple, réaliser un routage au niveau hardware sur un réseau de processeurs interconnectés par des *crossbars switches* adaptés, en faisant du multiplexage sur les liens physiques. Ce qui permet à l'utilisateur de considérer le réseau comme complètement connecté. Le fait d'utiliser d'office des techniques de routage amène une perte des performances de tels réseaux, mais les facilités amenées par un tel mécanisme sont immenses.

d) Communication inter-processeurs

Les communications interprocesseurs[3] sont très coûteuses et dans la plupart des applications réelles, elles prédominent sur le calcul proprement dit.

Le modèle le plus courant pour le coût d'une communication de L données entre deux voisins directs peut être représenté par : $\beta + \tau L$.

Le facteur β est le *start-up*, il peut être du même ordre de grandeur que τ (donc négligeable si L est grand), ou bien prédominant. τ est le taux de transmission, ou encore l'inverse de la bande passante du lien. Sur certaines machines, ce temps est considéré comme constant (τ négligé devant β), ou bien au contraire on suppose que l'on manipule de grands messages et alors on ne garde dans le modèle que le terme en τ.

On va supposer que des communications à partir d'un même processeur peuvent avoir lieu en parallèle.

Généralement, sur une machine munie d'un routeur, le temps pris par le mécanisme de routage est tel que le temps de communication peut être considéré comme constant, quel que soit l'éloignement des processeurs.

Il existe différentes procédures de communications permettant chacune d'atteindre un nombre divers de processeurs. On peut ainsi communiquer une donnée :

OTO (One To One): d'un processeur à un autre,

OTA (One To All): d'un processeur à tous les autres (diffusion),

ATO (All To One): de tous les processeurs à un seul d'entre eux (rapatriement),

ATA (All To All): de tous les processeurs à tous les autres (échange global).

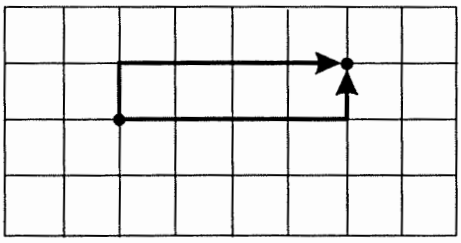
---

[3] Référence : cours de DEA d'algorithmique parallèle d'Yves Robert et de Denis Trystram, ENSIMAG 1990.

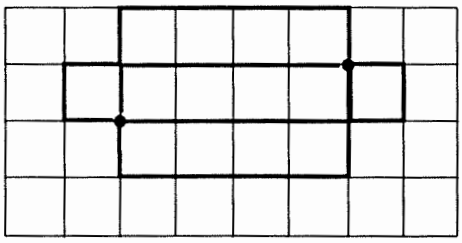
Dans les deux points suivants, nous allons voir le cas du OTO et du OTA.

(1) OTO

La communication OTO, d'un processeur à un autre (non forcément voisins directs) est assez simple en général. La technique la meilleure est basée sur la segmentation du message et le pipeline de l'envoi des paquets successifs. En général, on essayera d'utiliser au maximum l'hypothèse de l'utilisation simultanée des liens de façon à avoir le maximum de chemins possibles pour relier les deux processeurs. Le problème est ici de trouver des chemins de longueur égale. Prenons l'exemple d'une 2-grille (cf. figure 2.9). Dans le premier cas, un message de taille L/2 est envoyé sur chacun des deux chemins disjoints de longueur d, dans le second, 4 messages de taille L/4 sont envoyés le long de quatre chemins de longueur d+2.



Envoi en pipeline en utilisant 2 chemins sur une grille.



Envoi en utilisant 4 chemins.

Figure 2.10. OTO sur une grille.

(2) OTA (diffusion)

Soit un réseau G(Δ,D), de degré Δ et de diamètre D, sur lequel on désire diffuser un message de taille L. La stratégie "bête" (en glouton) peut être obtenue après D étapes de communications sur des messages de taille L, en suivant le plus long chemin sur un arbre de recouvrement, soit : $D(\beta+\tau L)$.

Une amélioration peut être apportée en découpant le message à envoyer en q paquets de même taille (méthode du pipeline).

Le temps d'une étape est $\beta+\tau\frac{L}{q}$, la distance la plus longue qui sépare deux processeurs quelconques est D. Soit au total (D+q-1) étapes :

$$(D+q-1)(\beta+\tau(L/q))=(D-1)\beta+\tau L+q\beta+\tau\frac{L(D-1)}{q}.$$

Il est facile de calculer le découpage optimal :

$$\beta-\tau\frac{L(D-1)}{q^2}=0$$

$$q_{opt}=\sqrt{\frac{\tau L(D-1)}{\beta}}$$

Ce qui correspond à un temps de diffusion de : $(\sqrt{(D-1)\beta}+\sqrt{\tau L})^2$.

La technique de pipeline nous a permis de gagner en gros un facteur D. Si l'on veut conjuguer l'effet pipeline avec des envois en parallèle sur les liens (Δ au plus), il faut trouver Δ arbres à arêtes disjointes (ce n'est pas toujours possible!).

Borne théorique :

Le temps minimum de diffusion d'un message de taille L est obtenue en D étapes de communication sur des messages élémentaires de taille la plus petite possible (1), plus le temps mis par le reste du message (soit L-1) arrivant en pipeline par tous les liens à la fois (en supposant idéalement que c'est possible ...):

$$t_{opt}(OTA)=D(\beta+\tau)+(\frac{L-1}{\Delta})\tau=D\beta+(\frac{L-1}{\Delta}+D)\tau$$

III. La synchronisation, la communication et l'exclusion mutuelle

Introduction. 27
Les sémaphores. 28
Définition 28
Utilisation 29
Implémentation 29
Les sémaphores faibles 29
Les sémaphores forts 30
Critique 30
Exemple 31
Les moniteurs 32
Définition 32
Implémentation 32
Critique 33
Exemple 34
Les messages 35
Définition 35
Nomination de l'émetteur et du récepteur 35
Nomination directe 35
Nomination globale 36
Les ports 36
Distinction synchrone/asynchrone 36
Définition 36
Asynchrone 37
Asynchrone/synchrone 37
Synchrone/asynchrone. 38
Synchrone 38
Les commandes gardées. 38
Critique 38
Exemple 39

A. Introduction.

Le but de ce chapitre est d'étudier les problèmes engendrés par la gestion d'un ensemble de processeurs et trois techniques classiques utilisées pour les résoudre.

Un programme séquentiel spécifie l'exécution séquentielle d'une liste d'instructions ; son exécution est appelée processus. Un programme concurrent spécifie deux ou plusieurs programmes séquentiels qui peuvent être exécutés de manière concurrente, comme des processus parallèles. Par abus de langage on appelle processus, les programmes séquentiels qui constituent les programmes parallèles. Dans le cas des MIMDs, on peut envisager que plusieurs processus parallèles se partagent un seul processeur à l'aide d'un mécanisme de partage du temps de ce processeur (*time sharing*).

Dans le but de coopérer, les processus s'exécutant de manière concurrente doivent communiquer et se synchroniser. La communication permet à l'exécution d'un processus d'influencer l'exécution d'un autre. La communication inter-processus est basée sur le passage de messages ou sur l'utilisation de variables partagées (variables se situant dans un espace mémoire commun). La synchronisation permet d'introduire un délai dans l'exécution d'un processus en attendant que certaines contraintes soient satisfaites : par exemple, dans le cas de deux processus communicants à l'aide de variables partagées, le processus lecteur doit attendre que l'autre processus ait bien écrit les variables.

Plusieurs processus ne peuvent accéder en même temps à certaines ressources (par exemple de la mémoire partagée) sous peine d'avoir des effets les plus variés et les plus désagréables (incohérence de données, ...). Il faut donc que, quand plusieurs processus désirent utiliser une même ressource non partageable, chaque processus puisse se réserver et utiliser la ressource sans interférence avec les autres processus, on appelle ce mécanisme exclusion mutuelle. Une section critique est une section du code d'un processeur qui ne peut pas être utilisée par plus d'un processus à la fois et dont l'exécution ne peut être interrompue, elle contient généralement le traitement d'une ressource partagée.

Un ensemble de processus est dans un *deadlock* quand ils attendent un événement qui n'aura jamais lieu. Soient deux processus P_1 et P_2, et deux ressources, x et y. Supposons

que le processus P_1 a un accès avec exclusion mutuelle à la ressource x et tente d'obtenir un accès à la ressource y. Si le processus P_2 a un accès avec exclusion mutuelle à la ressource y et tente d'obtenir un accès à la ressource x, les processus P_1 et P_2 provoqueront un *deadlock*.

Nous allons maintenant étudier trois des techniques[4], les plus utilisées, permettant de résoudre les problèmes de synchronisation, de communication et d'exclusion mutuelle : les sémaphores, les moniteurs et les messages.

B. Les sémaphores.

1. Définition

Un sémaphore est une variable entière non-négative à laquelle on peut accéder par deux opérations atomiques :

P(s) : Met en attente le processus appelant jusqu'au moment où $s>0$ et ensuite décrémente s de 1.

V(s) : Incrémente s de 1.

Les opérations P(s) et V(s) sont indivisibles, c'est-à-dire que deux processus ne peuvent exécuter en même temps une opération sur un même sémaphore. Si certains processus tentent d'accéder à un sémaphore pendant qu'une opération P(s) ou V(s) est en cours d'exécution sur celui-ci, ils sont mis en attente et les opérations demandées seront exécutées ultérieurement de manière arbitraire et séquentielle.

[4] Références bibliographiques : [Banatre 90], [Dinning 89], [Gehani 88], [Perrot 87], cours de 2ème licence en informatique, "OS matière approfondie", de J. Ramaekers, FUNDP 90.

2. Utilisation

Les sémaphores sont utilisés pour l'exclusion mutuelle et la synchronisation conditionnelle. Les sémaphores qui ont comme valeur possible 0 et 1 sont appelés sémaphores binaires et ceux qui peuvent prendre une valeur arbitraire, sémaphores généraux ou compteurs (*general or counting semaphores*), ceux-ci sont utiles quand il y a des instances multiples d'une même ressource partagée. Quand un sémaphore a une valeur supérieure à zéro il est appelé ouvert sinon il est dit fermé.

Pour permettre l'exclusion mutuelle, un sémaphore s est associé à une section critique et est utilisé pour garantir un accès séquentiel à celle-ci. Avant d'entrer dans la section critique, chaque processus doit exécuter P(s) ; avant d'en sortir, il doit exécuter V(s). Pour permettre la synchronisation conditionnelle, un sémaphore doit être associé avec chaque condition : un processus exécute V(s) sur un sémaphore quand il met la condition à vrai.

3. Implémentation

Les sémaphores sont classés comme étant faibles ou forts sur base de l'implémentation sous-jacente.

a) Les sémaphores faibles

Les sémaphores faibles correspondent à une attente active (*busy waiting*), c'est-à-dire que quand on exécute une opération P sur un sémaphore faible, un processus vérifie de manière répétée la valeur du sémaphore jusqu'au moment où il devient supérieur à zéro ; à ce moment, le processus décrémente le sémaphore, vérifie que cette décrémentation a une action légale, et continue. Une opération V(s) incrémente simplement le sémaphore. En général les sémaphores faibles ont des délais très courts mais gaspillent le processeur. De plus beaucoup d'implémentations de sémaphores faibles ne garantissent pas qu'un processus ne tourne indéfiniment quand d'autres processus entrent dans la section critique.

b) Les sémaphores forts

Les sémaphores forts sont une implémentation alternative utilisant une attente passive (*positive wake-up*). Pour assurer que les processus mis en attente entrent dans la section critique d'une manière FIFO (*first-in-first-out*) afin d'assurer une certaine équité entre les processus, les sémaphores forts utilisent des files d'attente pour la maintenance du set de processus bloqués. Si un processus exécutant un P(s) est mis en délai, il sera mis sur la file d'attente du sémaphore s. Sinon il ferme le sémaphore et entre dans la section critique. Si un processus exécute un V(s) et que la file d'attente de s n'est pas vide, on le signalera à un processus. Sinon il ouvre le sémaphore.

4. Critique

Le défaut principal des synchronisations basées sur les sémaphores est que la responsabilité du contrôle d'accès dépend du programmeur qui doit décider quand on synchronise et à quelles conditions. Les sémaphores ne fournissent pas de renfort automatique d'accès à l'exclusion mutuelle de variables partagées ; par erreur, un programmeur peut oublier d'inclure toutes les références aux variables partagées dans les sections critiques. Un P(s) ou un V(s) oublié peut provoquer rapidement un *deadlock*. Le code de synchronisation peut être réparti à travers tout le programme. A moins que le code ne soit précautionneusement structuré, les programmes utilisant des sémaphores sont très difficiles à comprendre, à construire et la preuve de correction de ces programmes est très difficile à établir.

5. Exemple

Nous allons reprendre l'exemple classique ([Banatre 90], [Perrot 87], ...) de gestion d'un tampon à l'aide de sémaphores. Ce tampon est accessible par deux processus : un producteur et un consommateur.

Nous introduisons trois sémaphores :

- **mutex** garantissant l'accès exclusif au tampon ;

- **libre** qui permet de suspendre le producteur lorsque le tampon est plein, **libre** est initialisé au nombre d'emplacements libres, c'est-à-dire **n**;

-**vide** qui permet de suspendre le consommateur lorsqu'il désire consommer dans un tampon vide.

Voici une solution possible :

```
début
var     char tampon[1..n];
sémaphore mutex(1), vide(0), libre(n);

processus producteur()
elt est un entier;
tantque (vrai )
        {
        elt:=produire; /* Production d'un élément */
        P(libre);
        P(mutex);
        ranger(elt,tampon);
        V(mutex);
        V(vide);
        };
fin producteur;
```

```
processus consommateur()
var entier elt;
tantque (vrai)
        {
        P(vide);
        P(mutex);
        elt: = ôter_un_élément(tampon);
        V(mutex);
        V(libre);
        consommer(elt)
        };
fin consommateur;
fin
```

C. Les moniteurs

1. Définition

Les moniteurs sont basés sur le concept d'encapsulation de l'état d'une ressource partagée avec les opérations qui la gèrent. Un moniteur est constitué d'un ensemble de variables représentant l'état d'une ressource partagée, les variables de condition, les variables temporaires, les procédures qui effectuent des opérations sur les variables, et les procédures d'initialisation. Les variables définies dans un moniteur peuvent être accédées seulement à partir des procédures de ce moniteur. Seulement un processus, à la fois, peut exécuter une procédure d'un moniteur. Donc l'exécution d'une procédure d'un moniteur garantit un accès en exclusion mutuelle à toutes les variables du moniteur.

2. Implémentation

La synchronisation conditionnelle est généralement fournie grâce à un mécanisme d'attente passive (*signal-wait*). Quand un processus attend une variable de condition, il abandonne le moniteur jusqu'au moment où il reçoit le signal. Quand un processus signale une variable de condition, il choisit un processus attendant la condition et l'active. Le moniteur reste alloué et le processus signalant est suspendu jusqu'au moment où le

processus nouvellement activé quitte le moniteur ou exécute une nouvelle action d'attente. Les processus en attente sont généralement gérés par un mécanisme de file d'attente.

3. Critique

Un moniteur, s'il est utilisé avec précaution, est un puissant outil de synchronisation. Les programmes utilisant des moniteurs sont facilement compréhensibles parce que toutes les opérations manipulant des objets partagés apparaissent dans la définition du moniteur ; l'utilisateur d'un moniteur peut ignorer les détails d'implémentation. L'écriture de programmes corrects est simplifiée par le fait qu'accéder des variables partagées est mutuellement exclusif, ce qui est une amélioration sur le contrôle ad hoc fourni par les sémaphores. La méthode d'Hoare de preuve axiomatique peut être appliquée aux programmes utilisant les moniteurs. Les préconditions et postconditions peuvent être associées avec chaque appel de procédure, la routine d'initialisation et les opérations signal et wait.

Le mécanisme de signal (*signal-wait*) accroît aussi la complexité de démonstration de correction du programme. Quand un processus exécute soit une opération *signal*, soit une opération *wait*, il abandonne le contrôle du moniteur pour permettre à un autre processus de s'exécuter. L'état du moniteur peut changer de manière drastique entre le temps où un processus relâche ou reprend le contrôle d'un moniteur. Cela peut rendre le mécanisme de signal difficile à utiliser correctement.Vu que les moniteurs inhibent complètement l'accès aux données partagées, ils peuvent être inadaptés à un environnement parallèle.

4. Exemple

L'exemple suivant montre comment un moniteur peut être utilisé pour implémenter des primitives producteur/consommateur mono-*buffer*. Le protocole de base est le suivant : quand le buffer est vide, tous les consommateurs doivent attendre ; quand il est plein, tous les producteurs doivent attendre ; et deux processus ne peuvent jamais accéder le *buffer* simultanément.

```
Producteur-consommateur : moniteur
{
        buffer est la variable partagée buffer;
        est_plein est un booléen;
        vide et plein sont des conditions;

        procédure produce( data )
        {
                si (est_plein) alors
                        attend( vide );
                buffer:=donnée;
                est_plein:= vrai;
                signal( plein );
        };

        procédure consume( data )
        {
                si (est_plein==faux) alors
                        attend( plein );
                donnée := buffer;
                est_plein:= faux;
                signal( vide );
        };

        initialisation
                est_plein = faux;
        fin
}
```

D. Les messages

1. Définition

Le passage de messages est une approche différente de la synchronisation. Dans le passage de messages, toute communication interprocessus doit prendre place à travers la transmission explicite de valeurs entre deux ou plusieurs processus, au lieu de lire ou d'écrire des variables partagées, les processus envoient ou reçoivent des messages. Traditionnellement le passage de message a été considéré seulement dans le cadre des systèmes distribués. De nos jours cependant, plusieurs systèmes multi-processeurs combinent le passage de messages avec les mémoires partagées.

L'implémentation du *send* et du *receive* ont deux paramètres de base : la manière dont les processus sources et destination sont nommés et la façon de synchroniser la communication.

2. Nomination de l'émetteur et du récepteur

Il y a trois méthodes possibles pour spécifier les désignateurs de la source et la destination (aussi connu comme étant les canaux de communication).

a) Nomination directe

La méthode la plus simple est la nomination directe (*direct naming*), dans laquelle les noms des processus source et destination sont utilisés comme désignateurs. Quelques procédés utilisés en programmation parallèle, tels que celui du pipeline (qui est constitué par n processus qui communiquent, tels que le $i^{ème}$ processus reçoit un message du $i\text{-}1^{ième}$ et l'envoie au $i+1^{ième}$), sont particulièrement désignés pour la nomination directe. Les autres, tels que le problème client/serveur (qui est constitué de multiples clients et de serveurs, avec chaque serveur servant n'importe quel client), sont moins bien servis par la

nomination directe car celle-ci ne permet pas à un processus de recevoir des messages d'un processus arbitrairement choisi parmi tous les émetteurs possibles.

b) Nomination globale

Une approche plus générale est la nomination globale ou boîtes aux lettres (*global naming or mailboxes*). Au lieu d'un nom de processus, une boîte aux lettres est spécifiée comme étant la source ou la destination d'un message. N'importe quel processus associé à une boîte aux lettres peut envoyer ou recevoir de celle-ci. Cependant le coût pour tous les membres d'une boîte aux lettres de maintien du statut courant peut être élevé.

c) Les ports

La troisième approche, les ports, combine nomination globale et directe. N'importe quel processus peut envoyer des messages à un port, mais seul un processus peut recevoir des données de celui-ci. Les ports permettent cette nomination anonyme via les boîtes aux lettres sans les frais traditionnels.

3. Distinction synchrone/asynchrone

a) Définition

Le second paramètre de base indique si le message est synchrone ou asynchrone.

Une opération est asynchrone si son exécution ne retarde jamais celui qui l'invoque. Tandis qu'une opération synchrone peut retarder son exécution tant qu'une condition spécifique n'est pas remplie (ex : j'attends d'envoyer mes informations tant que je n'ai pas un récepteur prêt à les recevoir).

Le point de vue synchrone/asynchrone peut être examiné au niveau de l'émetteur ou du récepteur. Nous pouvons caractériser une communication par un couple indiquant si l'envoi et la réception se font de manière synchrone ou asynchrone.

b) Asynchrone

L'utilisation de communications asynchrones/asynchrones impliquerait des pertes d'information. Cependant la notion de fort synchronisme, dont je reprends la définition de Banatre ([Banatre 90]) dans le paragraphe suivant, correspond à la définition de totalement asynchrone (pas de délais ni du côté émetteur ni du côté récepteur).

Depuis quelques années de nouveaux langages de programmation adaptés à la description de problèmes temps réels commencent à émerger. Ils sont tous fondés sur une hypothèse de synchronisme fort (appelé aussi réactivité) qui assure que l'exécution d'une action prend un temps nul et donc que la réaction à un évènement externe est instantanée. Cette hypothèse simplificatrice permet de rendre beaucoup plus aisés les raisonnements sur le temps. Par exemple, dans un formalisme habituel, exécuter deux instructions demandant un délai de 3 milli-secondes n'est pas équivalent à exécuter une instruction demandant un délai de 6 milli-secondes, l'analyse d'un instruction prenant du temps. Dans un formalisme fortement synchrone, cette équivalence devient vraie ... ce qui a le mérite de mieux correspondre à l'intuition.

En revanche, la mise en oeuvre de cette hypothèse se heurte à un problème sérieux : la machine utilisée doit être infiniment rapide.

Dans la réalité, il suffit que la vitesse de la machine effectuant les calculs soit nettement supérieure au délai de réaction attendu afin que l'observateur du système perçoive celui-ci comme réactif. Ce qui implique aussi l'utilisation de messages synchrones.

c) Asynchrone/synchrone

Une deuxième manière de procéder est de rendre l'envoi asynchrone et la réception synchrone. Ceci implique un espace de tampons sans limites ; en pratique, un nombre limité de tampons est utilisé et un émetteur échouera si ceux-ci sont tous pleins. Le principal défaut de cette méthode est la place mémoire prise par les tampons.

Ce mode peut être facilement transformé en mode synchrone : il suffit en effet de prévoir que le récepteur envoie un accusé de réception après la lecture d'un message et que l'émetteur attende cet accusé.

d) Synchrone/asynchrone.

Le mode envoi synchrone et réception asynchrone ne s'utilise pas à ma connaissance. Il serait toujours possible de trouver une situation particulière où cela serait utile ; mais ici, nous n'examinerons que les traitements généraux. De plus dans un système multitâches, il est aisé de transformer un type de communication en un autre.

e) Synchrone

Le type de communication où on émet et reçoit de manière synchrone est appelé rendez-vous. Ce mode permet de ne pas avoir de gestion de tampons (asynchrone/synchrone) et de ne pas avoir de perte d'information (asynchrone/asynchrone).

Transformer des communications synchrones en asynchrone/synchrone est plus complexe que le contraire. On peut par exemple, quand un processus désire émettre un message, créer un nouveau processus qui sera chargé d'émettre le message. Ce sera donc ce dernier qui sera bloqué plutôt que le processus initial.

4. Les commandes gardées.

Dijkstra a mis au point la notion de commande gardée, il s'agit d'une ou plusieurs instructions précédées d'un garde. Le garde est une expression booléenne. Le garde réussi si l'expression booléenne est évaluée à vrai et la ou les instructions qui le suivent sont exécutées ; il échoue si l'expression booléenne est fausse.

Plusieurs concepteurs de langages ont enrichi la notion de commandes gardées, en permettant de tester, par exemple, pour un processus récepteur, parmi plusieurs émetteurs lequel est prêt le premier à émettre, ce qui amène un comportement non-déterministe du programme. En effet, lors d'exécutions successives du programme, rien ne dit que ce sera toujours le même émetteur qui sera prêt en premier.

5. Critique

Comme les sémaphores, le passage de message est d'utilisation correcte difficile quand les opérations d'envoi et de réception sont disséminées dans le texte du programme.

En général, il n'y a pas de mécanisme simple, comme les pré- et postconditions utilisées avec les moniteurs, pour prouver formellement que le passage de messages est correct. Il existe cependant des ateliers logiciels permettant de détecter, entre autres, si dans un programme particulier on peut se retrouver confronté à des problèmes d'inter-blocages ([Zampuniéris 90]).

6. Exemple

Voyons ce que donne l'exemple d'un tampon rempli par un producteur et utilisé par un consommateur :

```
processus buffer;
        variables
                N, un entier, est la taille maximale du tampon;
                tampon est un vecteur contenant N éléments (de 0 à N-1);
                taille, un entier, contient la taille du tampon;
                tête, un entier, est un pointeur sur l'élément à lire du tampon;
                queue, un entier, est un pointeur sur le prochain élément libre;
        tête: = 0 ; queue: = 0 ; taille: = 0;
        tantque (vrai) faire
        {
        garde((taille < N) et (le producteur veut envoyer un élément))
        -->     { recevoir tampon[queue] du producteur;
                  taille: = taille + 1;
                  queue: = (queue + 1) modulo N;
                };
        garde((taille > 0) et (le consommateur demande un élément))
        -->     { envoyer tampon[tête] au consommateur;
                  taille: = taille-1;
                  tête: = (tête + 1) modulo N;
                };
        };
fin processus;
```

```
processus producteur;
        élément;
        tantque (pas terminé)
                {
                génère(élément);
                envoyer élément au buffer;
                };
fin processus;

processus consommateur;
        élément;
        tantque (pas terminé)
                recevoir élément de buffer;
                utilise(élément);
        fin tantque;
fin processus;
```

IV. Les transputers

Introduction. 42
Un langage : OCCAM 42
Présentation 42
Exemple 44
Le transputer T800 44
Présentation 44
Processeur d'entier 46
Unité flottante 46
Mémoire 47
Horloges 47
Liens 47
Le transputer T9000 49

A. Introduction.

Dans ce chapitre, nous allons étudier les Transputers. Ces processeurs d'Inmos sont caractérisés par quatre liens de communication permettant de former des réseaux et par un ensemble de possibilités *hardware*, telles que la notion de message et la gestion du pseudo-parallélisme par *time-slicing*. Ces différentes possibilités ont été conçues dans le but de faciliter l'implémentation d'un langage de haut niveau destiné aux transputers : OCCAM.

B. Un langage : OCCAM

1. Présentation

Occam[5] est un moine et philosophe Irlandais qui a vécu au moyen-âge dont la contribution la plus connue est le principe suivant :

"the principle that unnecessary complication should be avoided and that the simplest solution is always preferable, other things being equal."

Il a donné son nom à un langage de programmation basé, bien entendu, sur le principe que tout doit rester simple. Ce langage est lui-même issu des CSP (*Communicating Sequential Processes*) de Hoare [HOARE 85]. Occam est donc un langage conventionnel et relativement simple. Un programme écrit en Occam est constitué d'un ensemble de processus concurrents, communicants entre eux et avec les périphériques, de manière synchrone à l'aide de canaux. Un canal est une structure de donnée abstraite qui supporte deux opérations : on peut y lire ou y écrire des données. Un canal Occam est une connexion point-à-point : à tout canal correspond un processus émetteur et un récepteur. La

[5] Références bibliographiques : [Dewar 90], [Inmos 88], [Pountain 88].

communication est synchronisée et a donc lieu quand les processus émetteur et récepteur sont prêts à communiquer. Les données sont alors copiées du processus émetteur vers le processus récepteur et les deux processus peuvent continuer.

Occam est basé sur les trois processus suivants :

L'assignation : v := e /* assigne l'expression e à la variable v */

La lecture : c?x /* La variable x va prendre la valeur lue sur le canal c */

L'écriture : c!exp /* La valeur de exp va être écrite sur le canal c */

Les processus primitifs sont assemblés par construction grâce à :

SEQ : chaque composant est exécuté l'un après l'autre.

PAR : Les composants sont exécutés en parallèle.

ALT : Le premier composant prêt est exécuté, si plusieurs sont prêts en même temps, un de ceux-ci est arbitrairement exécuté.

Chaque construction est elle-même un processus et peut faire partie d'un autre processus.

Une construction basée sur l'alternative (ALT) est constituée d'un ensemble de conditions (constituant des processus) et de processus gardés par ces conditions. Une condition contenant une lecture sur un canal ne peut être vraie que s'il existe un processus prêt à émettre à l'autre bout du canal. L'instruction de lecture (s'il en existe une) et le processus gardé par la première condition qui sera évaluée à "vrai" seront exécutés.

Occam a servi de modèle de base pour les transputers. Les grands principes d'Occam tels que la notion de processus et de communication par canaux se retrouvent au niveau matériel dans le transputer.

2. Exemple

Dans cet exemple, deux processus, reliés par un canal, effectuent des calculs, se communiquent une valeur et continuent leurs calculs.

```
CHAN OF INT canal: /* Déclaration d'un canal */
PAR
        INT X:
        SEQ
                ... /* Calculs */
                canal!X /* On envoie la valeur de X par le canal 'canal' */
                ... /* Le process continue */
        INT Y:
        SEQ
                ... /* Calculs */
                /* On reçoit une valeur par le canal et on la stocke dans Y */
                canal?Y
                ... /* Le processus continue */
```

C. Le transputer T800

1. Présentation

Le Transputer[6] (TRANSistor-comPUTER) d'Inmos comprend une famille de micro-circuits 16- et 32-bits, capables d'opérer seuls ou comme composants d'un réseau parallèle de Transputers (MIMD à mémoire distribuée).

[6] Références bibliographiques : [Dewar 90], [Hirsh 90], [Hoare 85], [Inmos 88], [Pountain 90], [Trystram 90].

Inmos est une firme anglaise dont la maison mère est située à Bristol, elle appartient au groupe Thomson.

Chaque transputer T800 (la dernière génération commercialisée) est constitué d'un processeur d'entiers 32-bits RISC (*reduce instruction set computer*) qui opère à 10 Mips (millions d'instructions par seconde), d'une unité à virgule flottante de 64 bits qui permet de faire 2 Mflops (millions d'opérations en virgule flottante par seconde), de 4 kbytes de RAM statique, d'une interface mémoire de 32 bits, de quatre liens séries standards permettant le passage de messages aux autres Transputers à une vitesse pouvant aller jusqu'à 10 Mbits par seconde, ainsi que de deux horloges cycliques et d'un ordonnanceur.

Il faut signaler que Occam n'est pas l'assembleur des transputers, mais simplement un langage de haut niveau dont certains aspects se retrouvent au niveau du langage machine.

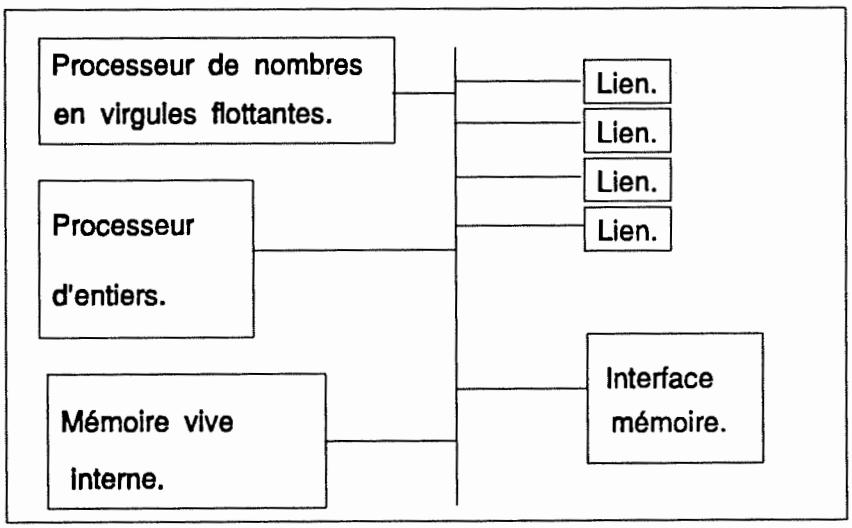


Figure 4.1. Architecture du T800.

2. Processeur d'entier

Le T800 ne possède qu'une pile de trois registres de calcul.

Les instructions du Transputer ont un format fixe sur 8 bits : 4 bits codent la commande suivis de 4 bits pour l'opérande. On a donc 16 instructions directes et des opérandes compris dans l'intervalle [0..15].

Cette limitation du nombre d'instructions et d'opérandes n'est qu'apparente car l'unité d'exécution a un fonctionnement particulier :

- Deux instructions particulières permettent de construire des opérandes signés 32 bits par étapes de 4 bits.

- Une des 16 instructions de base est en fait un point d'entrée pour plus de 140 instructions indirectes.

Le processeur principal du transputer est un processeur RISC, bien que se démarquant légèrement des processeurs RISC classiques : il possède un grand nombre d'instructions différentes, il ne possède pas de pipeline, il possède peu de registres. Mais les grands principes du RISC sont là : il ne possède pas de modes d'adressages compliqués, chaque instruction est codée sur 8 bits, chaque instruction de base (l'expérience nous montre qu'elles forment 70% du code d'un programme) peut ainsi être effectuée en un tick d'horloge.

3. Unité flottante

L'unité à virgules flottantes est le gros apport du T800 par rapport aux générations précédentes. Il s'agit d'une unité flottante scalaire 64 bits au format IEEE (le plus utilisé). La FPU (*Floating Point Unit*) fonctionne en concurrence avec le processeur d'entiers et dispose de sa propre pile de registres (3 registres 64 bits).

4. Mémoire

Les quatre kilo-octets de mémoire interne du T800 constituent les premières adresses de la mémoire ; cette mémoire, très rapide, fonctionne à la vitesse du processeur et peut se comparer aux registres internes des micro-processeurs usuels.

Une interface permet l'accès à une mémoire externe via un bus 32 bits de données et 32 bits d'adressage (4 giga-octets adressables). La mémoire externe est trois fois plus lente.

Les transputers qui ont été utilisés dans le cadre de ce travail possédaient entre 1 et 4 Mo de mémoire. Signalons aussi l'absence de dispositif matériel qui permettrait la mise en place d'une gestion paginée de la mémoire.

5. Horloges

De nombreuses possibilités offertes dans les autres ordinateurs au niveau du système d'exploitation se retrouvent ici au niveau matériel. En effet, un ordonnanceur *hardware* permet de faire du pseudo-parallélisme en utilisant les horloges.

Deux types de processus tournent sur un T800, ceux à priorité haute (pas de notion de découpage de temps) et ceux à priorité basse (au bout d'un certaine tranche de temps, le contrôle est passé au processus suivant). Les processus à priorité basse ne seront exécutés que si aucun processus à priorité haute n'est exécuté ou n'est prêt à être exécuté.

6. Liens

Les liens sont généralement utilisés pour former des réseaux de transputers. On peut soit les connecter liens à liens, soit utiliser des *crossbar switches* (qu'il soient simples ou qu'ils rééquilibrent les signaux).

Les liens peuvent opérer jusqu'à 10 Mbits par seconde.

Chaque transputer dispose de 4 liens bidirectionnels. Ces liens fonctionnent de manière autonome par l'intermédiaire d'un DMA (Direct Memory Access). A un lien physique sur un transputer correspondent deux canaux logiques.

Un canal est le moyen par lequel deux processus se synchronisent ou communiquent des informations (au moyen de rendez-vous). Le mécanisme de communication diffère selon que les deux processus se trouvent ou non sur le même transputer. Dans le premier cas, elle a lieu par le biais d'un transfert d'octets en mémoire centrale ; dans le second cas, par l'intermédiaire d'un lien série rapide.

Lors d'un transfert d'informations sur un lien physique (cf. figure 4.2), à chaque octet va correspondre un accusé de réception. Les envois d'octets et les accusés de réceptions sont multiplexés sur les liens. L'accusé de réception est codé sur deux bits comme l'indication d'envoi d'un octet (start bits). Ainsi s'il reçoit les bits 10 en entrée, le transputer sait qu'il s'agit de l'accusé de réception du dernier octet envoyé ; tandis que s'il reçoit 11, il sait que 8 bits de données vont suivre.

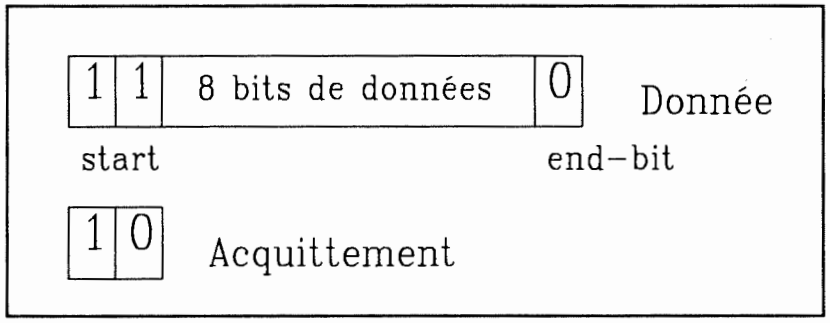


Figure 4.2. Protocole de transmission.

Notons aussi que la communication d'un réseau de transputers vers le monde extérieur (terminal, ...) s'effectue via un ou plusieurs liens physiques. Il suffit pour cela que, par exemple, on connecte à l'autre extrémité d'un lien d'un transputer, un ordinateur capable d'utiliser le protocole propre à ce lien.

D. Le transputer T9000

La génération suivante de transputers, le T9000 devrait permettre de décupler les débits des liens et les performances du processeur d'entier et de celui qui s'occupe des virgules flottantes.

Un ensemble de T9000 connectés avec le nouveau *crossbar switch* C104 pourra être vu comme un réseau complètement connecté. En effet le routage parmi les C104 et le multiplexage sur les liens physiques sera effectué de manière matérielle.

Le couple T9000, C104 devrait aussi permettre le routage aléatoire, c'est-à-dire permettre de générer automatiquement des routes différentes pour un même source et une même destination, cela afin d'éviter l'encombrement de certaines routes.

Signalons tout de même que les études déjà effectuées en ce qui concerne le routage et les communications en général sur les transputers seront encore d'un certaine utilité pour les programmes utilisant de manière intensive les communications inter-transputers. Utiliser deux routes pour transférer des informations sera toujours plus rapide que d'en utiliser une seule.

Le T9000 sera en outre doté d'un pipeline à 5 étages, ce qui confirme bien l'orientation RISC prise par Inmos.

V. Un réseau de transputers : le Méganode

Le projet ESPRIT 1085 52
Description de l'architecture 53
Configuration et utilisation du réseau en C3L 56
Modèle et extension du C 56
Configuration du réseau 57
Exécution d'un programme 60

A. Introduction.

Ce chapitre décrit un réseau de transputers appelé Méganode. Nous allons voir son origine, ses caractéristiques architecturales et les outils de base fournis avec celui-ci.

B. Le projet ESPRIT 1085

Le projet Esprit 1085 est un programme de la CEE qui a eu pour but le "Développement d'un ordinateur multi-processeur à grande performance et à faible coût". Les sept partenaires français et anglais étaient dirigés par le Royal Signals and Radars Establishment (RSRE) du ministère de la défense britannique. Chaque partenaire possédait des responsabilités directes pour des aspects spécifiques du programme.

. RSRE : Gestion globale du projet et développement d'applications de traitement du signal.

. Apsis : conception et fabrication assistée par ordinateur, y-compris l'implémentation d'un simulateur logique de multi-transputers.

. IMAG (Grenoble) : Les environnements de programmation de haut niveau.

. Inmos : La conception et la fabrication d'un nouveau transputer muni d'un processeur à virgules flottantes, le T800.

. Université de Southampton : Architecture et conception du système. Elaboration du *software* de base.

. Telmat : Fabrication de machines pour les partenaires. Synthèse d'image.

. Thorn Emi : Conception détaillée du système. Conception d'un sous-système de gestion d'entrées-sorties en temps réel. Implémentation de logiciel de tests du système.

. Université de Liverpool : Etude et implémentation prototype de bibliothèques numériques.

Comme l'énumération précédente nous le montre le projet insistait particulièrement sur la possibilité de réaliser des applications réalistes fonctionnant sur une machine multi-processeur. En effet, dans ce projet estimé à 100 années-homme, près de la moitié de l'effort est concentré sur les applications (pour la plupart numériques).

Le multi-processeur cible du projet devait pouvoir posséder jusqu'à 1024 processeurs et atteindre le Gigaflop (1000 mégaflops).

C. Description de l'architecture

Le Méganode[7], construit par la société TELMAT, est un multi-processeurs basé sur les principes architecturaux développés dans le cadre du projet Esprit 1085. On peut en trouver un des rares exemplaires au Laboratoire de Modélisation et de Calcul (LMC) de l'Institut National Polytechnique de Grenoble. C'est effectivement au sein de ce laboratoire qu'est utilisé l'environnement de programmation décrit dans les paragraphes qui suivent.

Le méganode (cf. figure 5.1) possède actuellement 128 T800 directement utilisables par l'utilisateur et peut être étendu à 256 T800. Ces transputers possèdent chacun un Méga-octets de mémoire vive. Un certain nombre de transputers de contrôle et de *crossbar switches* permettent de réaliser n'importe quel graphe de degré inférieur ou égal à quatre. Les T800 sont regroupés par groupe de 16 en T-Node possédant un *crossbar switch* RSRE 72x72. Un groupe de 2 T-Node forme un tandem. Le méganode possède 4 tandems. Les tandems sont reliés à l'aide d'*internode switches C004*. Ceux-ci rééquilibrent les communications, ce qui implique un certain coût de passage : on peut estimer le temps de passage 1,5 fois plus long que pour un *switch* RSRE. Dans de nombreux langages, on n'a cependant aucun moyen de spécifier quels transputers on veut utiliser et de ce fait, on ne sait pas optimiser les programmes en maximisant les communications intra-tandems.

[7] Références bibliographiques : [Adamo 90], [Touzene 90], [Waille 90].

Le modèle de communication est $t_{com}=\beta+\tau n$ (cf. [Touzene 90]) où β est le temps de startup, τ le temps de transfert d'un octet et n le nombre d'octets :

Dans le cas de communications intra-tandem, on a :

β = 4,85 micro-sec et τ = 1,125 micro-sec.

Dans le cas de communications inter-tandems, on a :

β = 5,61 micro-sec et τ = 2,2 micro-sec.

Au moment de la rédaction du présent paragraphe aucun système d'exploitation n'est installé sur le méganode, qui implique que le système ne peut être alloué qu'à un seul utilisateur à la fois.

Le premier processeur du méganode est relié par un lien à un autre transputer se trouvant sur une carte (fournie par Archipel), elle même reliée à un SUN4 chargé de gérer les entrées/sorties (terminal et disques). **Toutes les entrées/sorties entre le Méganode et l'extérieur doivent transiter par le transputer de la carte Archipel.**

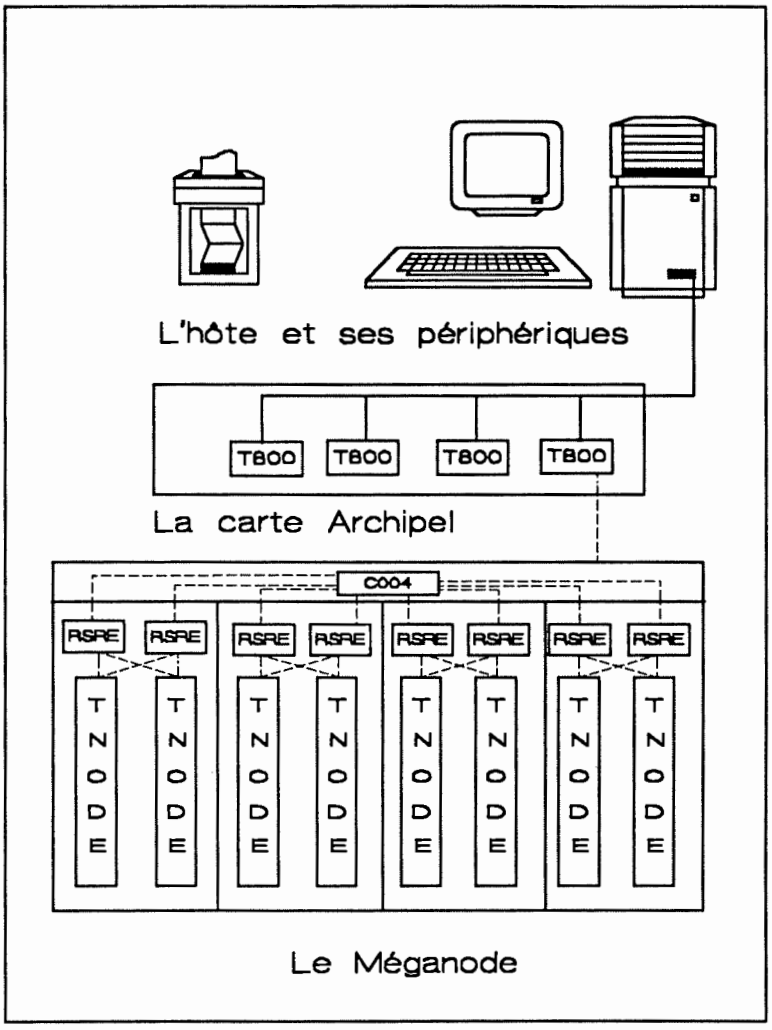


Figure 2.1. Le méganode et l'extérieur.

D. Configuration et utilisation du réseau en C3L

C3L, est un C destiné aux transputers et fourni par Telmat, constructeur du Méganode. Il s'agit du dernier C utilisé au sein du laboratoire LMC avant la mise au point de OUF, produit développé par le LMC et détaillé dans le chapitre suivant.

1. Modèle et extension du C

Le modèle de programmation en C3L est dérivé d'Occam.

L'unité de base est la tâche. Une tâche est constituée d'un ensemble de processus légers (avec time-slicing), appelés *threads*, qui sont exécutés sur un même transputer. Le fichier source d'une tâche ressemble à un programme C classique. Le point d'entrée de la tâche (main()) correspond au lancement d'un premier processus léger. Par la suite, il existe des primitives de lancement d'autres processus. L'ensemble des processus d'une même tâche se partagent un même espace mémoire.

Les *threads* des différentes tâches communiquent et se synchronisent à l'aide de canaux.

Sur le sun4 tourne un programme appelé serveur ('server'), fourni avec le C3L, qui n'a comme fonction que de recevoir des ordres (correspondant à des entrées/sorties).Ce programme va être chargé, par exemple, d'ouvrir un fichier, de lire une entrée au clavier, d'afficher des informations à l'écran, ... Le programme de l'utilisateur va envoyer les ordres au serveur grâce à une tâche située sur le processeur racine (situé sur la carte Archipel). Cette tâche est bien sûr connectée par des canaux au serveur.

C3L comprend la majeure partie des primitives et des bibliothèques classiques. Mais C3L n'est pas entièrement compatible ANSI, il lui manque entre autres la bibliothèque "stdarg.h" permettant d'avoir des fonctions comportant un nombre variable d'arguments.

Certaines bibliothèques lui ont été rajoutées afin de tirer parti des possibilités d'un réseau de transputers.

Ces bibliothèques comprennent la gestion des threads, les sémaphores, les communications synchrones (rendez-vous), les communications synchrones avec timer et la fonction alt. Cette dernière est une version limitée de l'alternative Occam. Elle permet de savoir parmi un ensemble de canaux en entrée lequel possède en premier un processus prêt à émettre de l'autre côté. On peut aussi retrouver dans ces différents langages la notion de canal logique : ainsi, au niveau du programme source, on ne verra pas la différence entre une communication s'effectuant entre deux tâches situées sur un même processeur d'une communication passant par un lien physique.

2. Configuration du réseau

Un programme C3L sur méganode est constitué de deux parties : le code pour un ensemble de tâches et un fichier de configuration.

Ce fichier contient une liste de processeurs que l'on veut utiliser, les connexions entre ces processeurs, une liste de tâche, une description des communication inter-tâches et le placement de tâches sur les processeurs.

Pour chaque tâche on précisera également quel est le fichier qui contient le code exécutable, le nombre de liens en entrée, le nombre de liens en sortie et l'occupation mémoire. Une seule tâche par processeur peut ne pas avoir de taille mémoire spécifiée, elle occupera l'espace restant libre sur le processeur.

Une information contenue dans le fichier de configuration qui ne correspondrait pas aux limites physiques des T800 entraînera une erreur lors de la configuration du réseau. La description à donner est assez contraignante car il faut explicitement citer tous les processeurs, toutes les tâches, les placements et les canaux. De plus le résultat obtenu n'est pas facilement lisible.

Exemple de fichier de configuration de C3L:

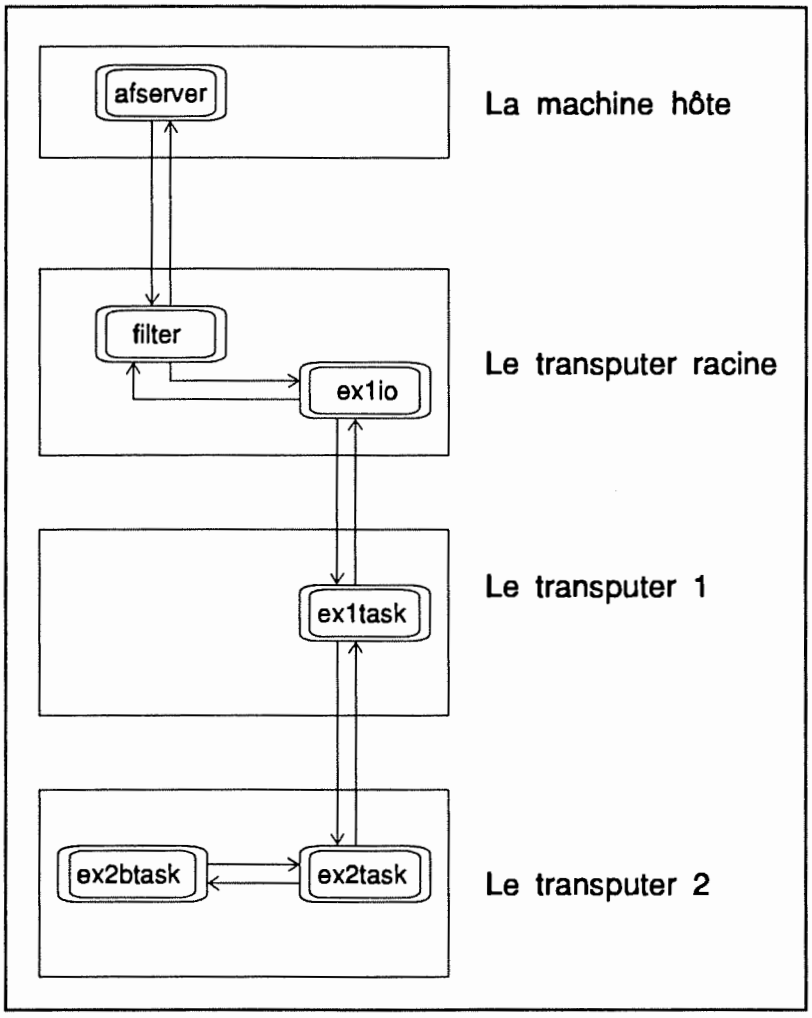


Figure 2.2. Exemple de configuration.

```
!!!!!!!!!!!!!!!
! Déclaration des processeurs utilisés
!!!!!!!!!!!!!!!

processor root /* transputer 0 de la carte Archipel */
processor proc1 /* un premier processeur du méganode */
processor proc2 /* un second processeur du méganode */
```

```
!!!!!!
! Définition des tâches :
! Chaque tâche est caractérisée par un nom, un nom de fichier,
! un nombre de canaux en entrées (ins),
! un nombre de canaux en sorties (outs) et
! une taille mémoire (data).
! La taille mémoire peut ne pas être spécifiée pour une tâche par T800.
!!!!!!
task afserver ins=1 outs=1 /* tâche serveur du Sun4 */
task filter ins=1 outs=1 data=10k /* tâche interface entre serveur et I/O */
task ex1io file=ex1io ins=2 outs=2 /* tâche utilisateur basée sur ex1io */
task ex1task file=ex1task ins=2 outs=2
task ex2task file=ex2task ins=2 outs=2
task ex2btask file=ex2btask ins=1outs=1 data=100k
!!!!!!
! Placement des tâches sur les différents processeurs
!!!!!!
place afserver host /* On place la tâche serveur sur le Sun4 */
place filter root
place ex1io1 root
place ex1task proc1
place ex2task proc2
place ex2btask proc2
!!!!!!
! Connexions inter-tâches : IN --> OUT
!!!!!!
connect ? filter[0] afserver[0] /* canal 0 (out) du filtre vers 0 (in) de afserver
connect ? afserver[0] filter[0] /* canal 0 (out) afserver vers 0 (in) de filter
connect ? ex1io[1] filter[1]
connect ? filter[1] ex1io[1]
connect ? ex1io[0] ex1task[0]
connect ? ex1task[0] ex1io[0]
connect ? ex1task[1] ex2task[0]
connect ? ex2task[0] ex1task[1]
connect ? ex2task[1] ex2btask[0]
connect ? ex2btask[0] ex2task[1]
!!!!!!
! Un canal bidon en entrée par tâche a été spécifié afin d'assigner
! un numéro de processeur logique à chaque tâche.
! La valeur de ce canal aura comme valeur le numéro de processeur.
!!!!!!
bind input ex1io[3] value=999
bind input ex1task[3] value=1
bind input ex2task[3] value=2
bind input ex2btask[2] value=2
!
! Wire connections ( Liens physiques )
!
wire ? host[0] root[0]
wire ? root[2] proc1[2]
wire ? proc1[0] proc2[0]
```

3. Exécution d'un programme

Après avoir compilé et linké les différentes tâches, on utilise un programme appelé *t_config* qui va créer un fichier contenant l'ensemble des processus destinés à tourner sur le réseau.

Ensuite, un programme appelé *t_switcher* permet de configurer la topologie du méganode suivant un fichier de configuration.

Et finalement un dernier programme va distribuer les différents processus utilisateurs sur le méganode.

On peut facilement imaginer les conflits qui peuvent résulter du jeu configuration/exécution quand deux utilisateurs essayent en même temps d'accéder au méganode ...

Notons qu'il existe deux manières de compiler un programme C pour réseau de transputers :

- Soit on utilise un *cross-compiler* qui permet de compiler un programme source sur un autre type de processeur, par exemple, le C de Logical System se compile sur SUN4.

- Soit on utilise les processeurs T800 situés sur la carte Archipel (c'est le cas en C3L) pour effectuer la compilation.

VI. Un environnement de développement C sur réseaux de transputers

Introduction 61
Un outil de description de réseau : Tn_Rita 62
Une extension de C : Ouf 64
Les tâches et les processus. 65
Les processus 65
Les canaux 66
L'horloge 67
Les sémaphores 67
Un debugger 67
Origine et présentation 67
Fonctionnalités attendues 68
Problèmes rencontrés 68
Solution proposée et fonctionnalités offertes 69
Implémentation 72
Améliorations 75

A. Introduction

La programmation du Méganode requiérait jusqu'à présent de la peine et du temps.

L'absence d'une file d'attente devant le Méganode engendrait des conflits entre utilisateurs : entre autres, la possibilité d'essayer d'exécuter un programme sur un réseau qui vient d'être reconfiguré par un autre utilisateur, ... De plus il fallait continuellement réessayer d'envoyer le programme à exécuter en espérant que le Méganode se libère.

La description de la configuration du réseau était un travail de longue haleine. En effet, imaginez la description à faire pour un programme parallèle composé de deux cents tâches utilisateurs. Le côté répétitif du travail était synonyme de frustration (dans un programme parallèle, de nombreuses tâches sont identiques), même avec les possibilités de copie de blocs des éditeurs de texte actuels. De plus le fichier de configuration, de par sa taille, n'est pas très lisible ni réutilisable.

Le programmeur se perdait parmi la propagation des C (d'abord Logical C, ensuite C3L et bientôt Inmos C) : se sentant obligé de s'adapter à la nouvelle version vu les possibilité offertes (C3L a introduit la notion de tâches, le C d'Inmos est compatible ANSI, ...), il perdait les bibliothèques de programmes déjà développées. De plus les modèles auxquels se réfèrent les différents C sont variables : existence de plusieurs niveaux de processus, ...

Corriger un programme n'était pas chose aisée. En effet comment savoir quelle était la valeur de telle variable dans telle tâche sur tel processus? Comment savoir si une communication entre deux tâches avait bien eu lieu et si la valeur passée était correcte? Seules les tâches placées sur le processeur racine et connectées via le filtre au serveur avaient cette possibilité. Dans les autres cas, l'utilisateur devait lui même organiser le routage des informations vers le processeur racine. Il est facile d'imaginer les difficultés rencontrées et la perte de temps engendrée par la mise en place de ces mécanismes.

Pour améliorer le quotidien des programmeurs et remédier aux faiblesses des outils de base fournis avec le Méganode, le laboratoire LMC a conçu et réalisé une série d'outils : une file d'attente pour le Méganode, un outil de description de réseaux de transputers (Tn_Rita), une interface abstraite (OUF) reprenant les différentes améliorations apportées au C par les C parallèles et un debugger couplé à cette interface. Nous allons étudier ces différents outils dans ce chapitre.

B. Un outil de description de réseau : Tn_Rita

Ecrire le fichier de configuration du réseau est une tâche ardue. Pour faciliter cette étape, il serait bon de disposer d'un langage disposant de possibilités algorithmiques et de description de réseau. Tn_Rita répond à ces demandes.

Tn_Rita est en fait une bibliothèque C, qui permet à l'utilisateur, grâce aux possibilités du C, d'effectuer plus rapidement sa description de réseau, en utilisant, par exemple les boucles.

Tn_Rita possède des primitives classiques de déclaration de processeurs, de tâches, de placement de tâches et de création de connexions inter-tâches. Tn_Rita possède en outre, les primitives opposées permettant d'annuler une déclaration ou une connexion. Ce qui permet d'obtenir des programmes lisibles : on connaît les algorithmes permettant d'obtenir les architectures classiques. Notons aussi qu'un processeur ou une tâche peut correspondre à une variable indicée.

Exemple d'utilisation de Tn_Rita :

Soit à configurer le Méganode de manière à ce que ses transputers soient reliés en un réseau linéaire de longueur np.

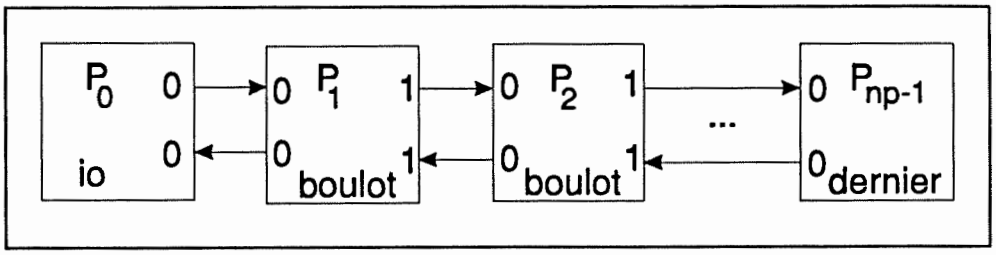


Figure 6.1. Réseau linéaire

Le programme descripteur du réseau est un programme C standard utilisant les fonctions de description de réseau. Un exemple possible réalisant le schéma de la figure 6.1 est donné par le fichier essai.c qui :

- déclare et instantie processeurs et tâches,

- affecte la tâche donnée "io.c" pour les entrées/sorties avec un lien en entrée et un en sortie,

- affecte la tâche donnée par "boulot.c" aux processeurs intermédiaires avec deux liens aussi bien en entrée qu'en sortie,

- affecte la tâche donnée par "dernier.c" au dernier processeur du réseau,

- crée les connexions comme indiqué sur la figure en utilisant les ports 0 (en amont des processeurs) ou 1 (en aval des processeurs).

Notons que les processeurs ont un numéro virtuel : l'indice i qui leur est attaché aux appels de DefProc. A la tâche "io" est de même associé la valeur 999.

Essai.c ;

```
#include "makecfg.c" /* il s'agit d'inclure Tn_Rita */

#define np 5
#define nbtach 5

main ()
      {
      Proc p[np]; /* Déclaration de np processeurs */
      Task io,T[nbtach]; /* Déclaration de la tâche io et de nbtach autres */
      int i; /* variable de travail */

      init_cfg(); /* initialisation de Tn_Rita */

      for (i=0 ; i<np ; i++) P[i]=DefProc(i); /*instanciation des processeurs */
      io=DefTaskIO("io",1,1);          /* instanciation de la tâche io, 1 canal en */
                                       /* et 1 en sortie. */
      BindTask(io,999); /* le numéro de la tâche io est 999 */

      for (i=0 ; i<=nbtach-2 ; i++) /* instanciation des tâches "boulot" */
            T[i]=DefTask("boulot",P[i],2,2);

      T[nbtach-1]=DefTask("dernier",P[nbtach-1],1,1); /* dernière tâche */

      DefBilink(io,0,T[0],0);    /* Déclaration d'un lien bi-directionnel */
                                 /* entre io et T[0] */

      for (i=0 ; i<(nbtach-1) ; i++) /* Déclarations de liens bi-directionnels */
            DefBilink(T[i],1,T[i+1],0);
      imp_cfg(); /* Appel à Tn_Rita */
      }
```

C. Une extension de C : Ouf

La prolifération des langages C (C3L, Logical C, Inmos C, ...) pour transputers peut poser certains problèmes, en effet, quelle est la portabilité d'un programme écrit dans un C non-standard ? De plus doit-on forcer tous les programmeurs à apprendre le nom donné dans chaque C à des primitives ayant des effets similaires ? Cette réflexion a amené l'idée de concevoir Ouf, une interface abstraite munie d'un debugger.

Ouf reprend toutes les extensions classiques apportées au C pour gérer le parallélisme: notion de tâches, de processus, de communications inter-processus et la notion de sémaphores. Ces instructions sont pour la plupart des macro-instructions qui vont transformer une primitive Ouf en une ou plusieurs instructions du langage cible. Ouf est disponible sous la forme d'une bibliothèque C.

L'idée d'utiliser une bibliothèque C pourra permettre de passer d'une manière non-contraignante pour l'utilisateur d'une interface abstraite à une extension particulière du langage C. Cela se fera en déterminant quels sont les critères permettant d'optimiser les instructions de synchronisation et de communication.

1. Les tâches et les processus.

La notion de tâche retenue pour OUF est identique à celle proposée par C3L.

La description d'une tâche en OUF est explicite : le début de la tâche est signalé par BEGIN_TASK("taskname") qui permet d'assigner un nom à la tâche. La fin de tâche est signalée par END_TASK.

Une tâche est placée sur un seul transputer, elle occupe une certaine partie de la mémoire de ce transputer et elle comprend un série de processus s'exécutant de manière concurrente. Les variables déclarées de manière globale sont partagées par l'ensemble des processus.

Le début d'une tâche correspond au lancement d'un processus léger (avec time-slicing).

2. Les processus

OUF permet aux tâches de lancer dynamiquement des processus. Un processus correspond à l'exécution d'une fonction qui ne renvoie pas de résultat (type "void").

Il est à noter que lorsque deux processus qui s'exécutent en parallèle utilisent une ressource partagée (écriture ou lecture sur un même canal par exemple, ou écriture à une même adresse mémoire, etc.) l'exclusion mutuelle n'est pas garantie, et doit être assurée

par l'utilisateur. L'exclusion mutuelle peut être garantie soit en inhibant ou en imposant des contraintes de précédence sur les processeurs entrant en section critique, soit en utilisant des sémaphores OUF.

Un processus en OUF peut lancer des processus en précisant que ces processus sont ses fils (notion de *fork* en Unix) et peut par la suite attendre la fin de l'exécution de l'ensemble de ses fils (notion de *join*).

3. Les canaux

Les communications s'effectuent selon le principe du rendez-vous. Un canal en OUF comporte une tâche émettrice et une tâche réceptrice.

Il existe trois types de canaux :

- Les canaux inter-tâches : ils sont déclarés dans le programme de configuration du réseau (Tn_Rita). Ces canaux lient deux tâches situées sur un même transputer ou sur deux transputers inter-connectés. A un lien physique ne peut correspondre qu'un canal dans chacune des deux directions, sous peine de générer une erreur lors de la configuration du réseau.

- Les canaux internes à une tâche : ils sont déclarés explicitement par l'utilisateur dans le programme décrivant la tâche. Ils permettent aux processus d'une même tâche de communiquer entre eux. Ces canaux ne sont pas accessibles par les autres tâches.

- lien physique : A chaque lien physique correspondent automatiquement deux canaux uni-directionnels (un en entrée et un en sortie) dont les ports d'accès sont situés à des emplacements mémoire fixes. Il s'agit ici de déconseiller l'utilisation directe des liens des transputers. Ces manipulations doivent rester exceptionnelles.

OUF possède des primitives de communication synchrone simples et d'autres avec timer (elle renvoient une valeur indiquant si la communication a échoué ou réussi dans la limite de temps spécifiée).

Nous venons de voir que les canaux lient des tâches. Il peut donc y avoir plusieurs processus émetteurs ou récepteurs pour un même canal. L'utilisation de cette possibilité sans précautions peut amener des comportements imprévisibles du programme. Pour éviter cela, il faut protéger de tels canaux à l'aide de sémaphores.

L'alternative OUF permet de connaître parmi une liste de canaux en réception lequel possède un processus prêt à émettre de l'autre côté.

Deux types d'alternatives sont possibles : blocantes ou non-blocantes.

4. L'horloge

OUF permet à un processus d'attendre un nombre de ticks d'horloge donné (*wait*) ou de connaître la valeur de l'horloge.

5. Les sémaphores

OUF possède une gestion de sémaphores généraux. Notons qu'il existe une primitive SemPOK(Sem s) qui teste l'état du sémaphore s : si le sémaphore est non-nul, le processus courant le prend, et la fonction renvoie une valeur non-nulle (équivalente à P(s)) ; sinon, la fonction renvoie 0.

D. Un debugger

1. Origine et présentation

C3L et Logical C ne possèdent pas de debugger. L'idée d'adjoindre un debugger à OUF fut judicieuse car elle permettait de compléter l'environnement de programmation offert aux utilisateurs du méganode. De plus le fait de concevoir et d'implémenter ce debugger m'a permis de concrétiser mes connaissances dans le domaine des environnements distribués.

Sur une machine séquentielle, quand un programmeur ne dispose pas de debugger, il peut toujours placer dans son programme des instructions envoyant des informations vers l'écran afin de procéder à certains tests. Sur le Méganode, le principal problème rencontré par les programmeurs était dû à son architecture. En effet, le Méganode est un réseau de transputers T800 couplé par l'intermédiaire d'un seul de ses transputers à un périphérique d'entrée/sortie, ce qui empêche l'envoi direct d'informations d'un transputer (autre que le transputer racine) vers la sortie (écran, disque, imprimante,...).

Le debugger a permis de diminuer fortement le temps de test des programmes utilisateurs.

L'utilisation d'un debugger ne dispense cependant pas le programmeur de l'emploi des techniques de génie logiciel pour le *programming in the large* et des techniques de preuve de programme pour le *programming in the small.*

2. Fonctionnalités attendues

Après avoir discuté avec l'équipe du laboratoire LMC, nous avons fixé les fonctionnalités du debugger comme suit :

- Le debugger doit pouvoir envoyer à l'utilisateur des indications concernant l'état du programme à un moment donné.

- Le debugger doit être capable d'enregistrer les communications qui ont lieu entre les tâches.

3. Problèmes rencontrés

Réaliser un debugger pour OUF a impliqué toute une série de choix à opérer.

Il y avait un premier impératif de temps, plusieurs personnes devant dès à présent travailler sur le Méganode, celles-ci avaient besoin d'avoir à leur disposition, rapidement, un environnement de travail agréable et surtout utile.

Le transputer en lui-même impose certaines contraintes : il ne possède que quatre liens de communications extérieurs, ce qui fait qu'on ne peut se permettre d'interdire à l'utilisateur l'utilisation d'un ou plusieurs liens qui serviraient au passage des informations propres au debugger.

On ne dispose sur le méganode que d'un méga-octets de mémoire par transputer, ce qui limite la taille du debugger et de ses données : il faut laisser suffisamment de place au programme utilisateur.

Un réseau de transputers ne possède pas de dispositif de routage (au niveau matériel), de plus un seul transputer est relié à l'extérieur (le transputer racine).

Un programme parallèle peut facilement être perturbé : vu qu'il existe des instructions permettant d'attendre un message du premier transputer prêt à émettre parmi plusieurs, on peut en modifiant la charge de travail des transputers changer l'ordre de déclenchement .

4. Solution proposée et fonctionnalités offertes

Pour réaliser un debugger inter-actif, il aurait fallu que des informations de debugging puissent s'échanger entre le transputer racine et tout autre transputer, il faudrait alors implémenter sur chaque transputer des tâches de routage permettant le multiplexage sur les liens physiques (vu le nombre limité de ceux-ci) et le routage des informations à afficher vers la machine hôte, et rajouter un entête à toutes les communications qui passent par les tâches de routage, sans quoi, le système ne saurait distinguer les communications "utilisateurs" des informations de debugging. Cette solution aurait amené un surcoût très important des communications, c'est pourquoi nous l'avons abandonnée.

La solution retenue fut de fournir à OUF un debugger post-mortem, ou plutôt un debugger qui remonterait ses informations à l'occasion d'un arrêt généralisé du programme. Par arrêt généralisé du programme, j'entends que tous les processus arrêtent momentanément leur exécution pour permettre au debugger d'envoyer vers la sortie (écran et disque) les informations collectées lors de l'exécution du programme utilisateur.

Le debugger est lui-même un programme parallèle, c'est-à-dire qu'il prend de la place dans la mémoire des différents transputers (200 Ko) et qu'il utilise du temps de calcul et de communication. Il faut que le programmeur en tienne compte, par exemple en augmentant la valeur des différents timers et en ne se fiant pas totalement à l'ordre de déclenchement des lectures avec gardes (instruction alt).

Quelles sont les primitives offertes par le debugger ?

deb_break() : permet de réaliser un point de synchronisation.

L'exécution de tous les processus doit soit être terminée soit être arrêtée grâce à l'appel de deb_break() pour que le debugger puisse envoyer au terminal les données à enregistrer.

deb_com : indique au debugger que les communications qui auront lieu par la suite dans cette tâche devront être enregistrées.

deb_end_com : : indique au debugger de ne pas enregistrer les communications qui vont suivre.

deb_writeX(format,arg1,..,argX) : envoi d'informations au debugger. Celles-ci seront enregistrées dans les fichiers de debugging. Les arguments sont semblables à ceux d'un printf classique. Le X suivant deb_write correspond au nombre de args.

Exemples :

```
deb_write("I Am at this step of the task");

deb_write1("y vaut %d",y);

deb_write2("%s --> %d",c,y);
```

deb_printfX(format,arg1,..,argX) : envoi d'informations au debugger. Celles-ci seront affichées à l'écran lors du prochain point de synchronisation. Les arguments sont semblables à ceux d'un printf classique. Le X suivant deb_printf correspond au nombre de args.

Le debugger vous fournit une série d'indications à l'écran sur l'état de votre programme : démarrage ; arrivée à un point de synchronisation et ouverture d'un fichier de debugging ; fermeture d'un fichier de debugging et redémarrage.

Les fichiers fournis par le debugger contiennent des tableaux semblables à celui-ci :

```
Enregistrements effectués sur le processeur racine :

Proc | task      |type|chan| informations
-----+-----------+----+----+-------------
  999| nom_tache|RB  |   0| 12
  999| nom_tache|info|    | Hi World
  999| nom_tache|RBt |   0| 13
  999| nom_tache|RWt |   0| 9876

Enregistrements effectués sur le Meganode :

Proc | task      |type|chan| informations
-----+-----------+----+----+-------------
    1|   ex1task|WW  |   1| 234
    1|   ex1task|info|    | I Am at this step
    1|   ex1task|WB  |   0| 12
    1|   ex1task|WBt |   1| 13
    1|   ex1task|WWt |   1| 9876
    1|   ex1task|WBt |   0| 13
    1|   ex1task|WWt |   0| 9876
    2|   ex2task|RW  |   0| 234
    2|   ex2task|RBt |   0| 13
    2|   ex2task|RWt |   0| 9876
    2|   ex2task|RB  |   2| 245
    2|  ex2btask|WB  |   0| 245
```

La première colonne contient le numéro du processeur où l'information a été collectée (numéro assigné par l'intermédiaire de Tn-Rita).
La deuxième colonne contient le nom de la tâche.
La troisième colonne contient le type d'information :

- soit inf pour une information enregistrée par deb_write.
- soit ABC, dans le cas d'une communication,

où A vaut R(Read) ou W(Write) ;
B vaut B(Byte), W(Word) ;
C vaut t s'il s'agit d'une communication avec timer.

La quatrième colonne indique, dans le cas d'un communication, le numéro de canal logique (Tn-Rita) ou 255 s'il s'agit d'une communication interne à la tâche.

5. Implémentation

Le debugger est écrit en OUF, ce qui facilite sa portabilité.

Deux possibilités nous étaient offertes, soit poster un petit enregistreur sur chaque tâche, soit en poster un par transputer. L'impératif mémoire nous a fait opter pour la seconde solution.

La modification de Tn_Rita, outil de description de réseau de transputers, nous a permis de relier chaque tâche "utilisateur" à la tâche "debugger" située sur le même transputer (cf. figure 6.2).

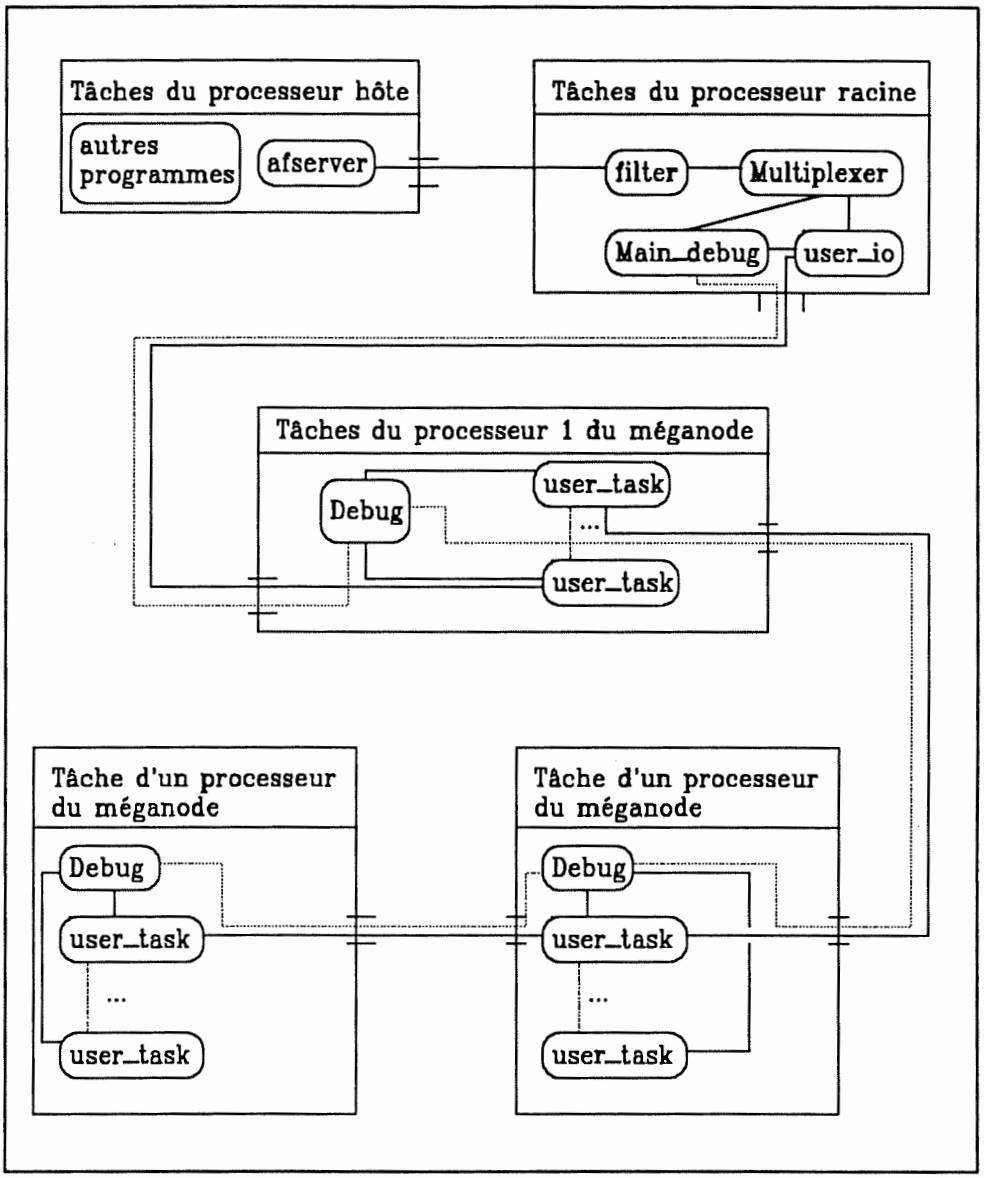


Figure 6.2. Exemple d'interconnexion des tâches utilisateurs et debugger.

Aucune communication n'est en cours lors d'un arrêt du programme, ce qui permet au debugger d'utiliser les liens de communications physiques puisque ceux-ci sont libres.

Pour permettre à l'utilisateur d'enregistrer les communications effectuées, il fallu détourner les primitives de communications de OUF de manière à envoyer un double de la communication vers l'enregistreur placé sur le transputer concerné.

Maintenant, comment savoir la topologie du réseau afin de pouvoir effectuer la remontée des informations vers l'hôte lors des points de synchronisation (arrêts généralisés)? La solution retenue comporte la recherche de la topologie sous la forme d'un arbre (en ne prenant pas en compte certains liens et en sachant que tout lien physique est bi-directionnel). Au démarrage du programme, il faut que toutes les tâches debugger essayent de rentrer en communication afin qu'il y ait une "connaissance" distribuée de l'arbre des transputers. Afin que lors de la remontée des informations chaque tâche debugger sache quel est sa mère et quelles sont ses filles.

Cela peut se faire de la manière suivante (cf. figure 6.3):

- Chaque tâche de debugging (excepté celle située sur le transputer racine) attend une information de sa mère (sa mère est la première tâche à lui envoyer une demande de maternité). Quand elle a reçu ce signe de vie, elle sait sur quel lien sa mère est connectée.
- Ensuite elle envoie un message de demande de maternité en parallèle sur ses trois autres liens et attend leurs réponses.
- La tâche debugger du processeur racine lance la première demande maternité.

Cette solution a l'avantage de fournir un arbre intéressant (presqu'équilibré) simplement grâce au fait que la recherche des branches s'effectue en parallèle.

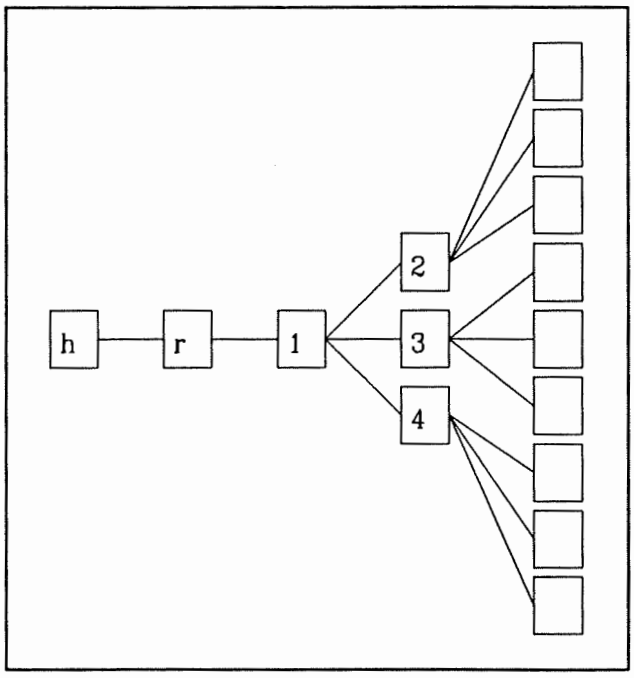


Figure 6.3. Recherche de l'arbre des transputers.

Les tâches debugger doivent enregistrer les informations que l'utilisateur veut collecter, tenir la comptabilité des tâches en vie, mortes et arrêtées se situant sur le même transputer. Lors de l'arrêt de toutes les tâches se situant sur leur processeur, elles doivent envoyer leurs informations à la tâche debugger mère et faire passer les informations venant de ses filles à sa mère.

OUF se comportera différemment lors de l'exécution d'un programme avec l'option de debugging. Dans un premier temps au début d'un tâche (BEGIN_TASK), il se mettra en attente de l'ordre de démarrage provenant de la tâche debugger située sur le même transputer. Ses instructions de communications devront envoyer si l'utilisateur le désire (primitive *deb_com*) un double à la tâche debugger. De plus Ouf va initialiser un *processus* qui la plupart du temps sera en attente sur deux canaux : le canal venant de la tâche debugger et un canal permettant le signal de break. Le seul type d'information susceptible de provenir de la tâche debugger est une information de redémarrage des tâches utilisateurs. Cela le *processus* peut le faire car il possède une liste chaînée des canaux sur lesquels les processus arrêtés sont en attente. Les éléments de cette liste lui ont été fournis directement par les processus considérés en utilisant le canal de signal de break précité (l'accès à ce canal est géré par un sémaphore).

6. Améliorations

Les programmes utilisateurs ont un comportement non-déterministes qui est dû, entre autres, à l'utilisation des alternatives : en effet, par exemple si deux processus sont sur le même tandem à une première exécution, rien ne garanti qu'à l'exécution suivante, ils ne soient pas sur un tandem différents et aient des temps de communication plus long, ce qui amène une non-répétabilité de l'exécution des programmes qui utilisent des alternatives. Une amélioration possible serait d'enregistrer l'ordre de déclenchement des différentes alternatives jalonnant le programme afin de répéter cet ordre lors de l'exécution suivante.

Une amélioration majeure serait aussi de permettre une phase inter-active lors d'un arrêt: l'utilisateur pourrait alors explorer les valeurs des variables des différentes tâches et même les modifier s'il le désire.

Une interface graphique pour le debugger est aussi envisagée.

VII. Programmation d'un réseau de transputer

Introduction 77
Qualité d'un algorithme parallèle 78
Contrôle du parallélisme 80
Ferme de processeur 80
Principe 80
Technique d'implémentation 81
Exemple 82
Parallélisme géométrique 82
Principe 82
Technique d'implémentation 83
Exemple 83
Description 83
Programme en pseudo-langage 85
Programme en OUF 86
Complexité de l'algorithme 88
Parallélisme algorithmique 89

A. Introduction

Le but de ce chapitre est simplement de montrer quelques aspects de la programmation d'un réseau de transputers. Il ne s'agit nullement de faire ici un cours de complexité ou d'algorithmique parallèle.

Dans les chapitre précédents, nous avons vu ce qu'était un réseau de transputers, quels étaient certains des outils dont on pouvait disposer pour programmer ce genre d'architecture. Mais comment réaliser des algorithmes parallèles et comment évaluer leur pertinence? La réponse à cette question n'est pas simple.

En fait actuellement, les spécialistes du parallélisme traitent problème par problème. Les différentes publications sur la solution de tel ou tel problème sur tel type d'architecture attestent du caractère prospectif de la programmation parallèle. On peut toutefois remarquer que parmi tout ce qui traite du parallélisme, on peut trouver certaines récurrences. En ce qui concerne le contrôle du parallélisme dans les environnements distribués, on retrouve trois grandes classes[8] : le parallélisme du type maître/esclaves appelé aussi ferme de processeurs, le parallélisme géométrique (le plus intéressant) et le parallélisme algorithmique. Le programmeur qui a pris connaissance de ces classes devrait pouvoir exprimer une partie importante des algorithmes qu'il doit implémenter à l'aide de ces paradigmes.

[8] Références bibliographiques : [Capon 90], [Hey 89], [Robert 90].

B. Qualité d'un algorithme parallèle

L'évaluation d'un algorithme parallèle n'est pas simple. Dans le cadre du calcul séquentiel, le modèle considéré est la machine de Turing. Le modèle présenté ici, RAM (Random Access Machine) est une adaptation de ce modèle théorique[9].

Deux caractéristiques sont considérées pour évaluer la qualité d'un algorithme séquentiel A_n :

- le temps (noté T_S): défini comme le nombre d'opérations effectuées sur des données bornées (ex: opérations sur des entiers, lecture/écriture d'un mot,...) ;

- la surface (notée S_S): définie comme le nombre de places en mémoire nécessaires à l'exécution de l'algorithme.

Par analogie, dans le cadre du calcul parallèle, la qualité d'un algorithme parallèle A_n est basée sur deux caractéristiques :

- la surface : définie comme le nombre maximum d'unités de calcul indépendantes (notée $H_{//}$) qu'il est possible d'utiliser pour tirer au mieux parti du parallélisme de l'algorithme. Nous considérons ici que l'on peut réutiliser un même processeur à différentes étapes successives de l'algorithme parallèle.

- le temps (noté $T_{//}$) : c'est le minimum des temps d'exécution de l'algorithme parallèle et peut être obtenu avec $H_{//}$ processeurs.

---

[9] Référence : cours de maîtrise (bac+4), "Algorithmique parallèle" de J-L Roch, Université Joseph Fourier (Grenoble).

De façon à pouvoir évaluer de manière précise le temps d'un algorithme parallèle à partir de sa description, il est fondamental de supposer que les processeurs travaillent en mode synchrone : cette hypothèse nécessaire mais invalidée en pratique n'est pas trop restrictive dans le cas où les unités de calcul sont physiquement de même nature (c'est le cas d'un réseau de transputers).

On appelle travail d'un algorithme parallèle $A_{//}$ la quantité notée $W_{//}$ définie par :

$$W_{//}=H_{//}*T_{//}$$

Le travail d'un algorithme séquentiel est défini par :

$$W_S=T_S$$

Soit A_S le meilleur algorithme séquentiel connu résolvant un problème P, et $A_{//}$ un algorithme parallèle résolvant P.

L'efficacité de l'algorithme $A_{//}$ est définie comme le rapport du travail A_S sur le travail de $A_{//}$:

$$e(A_{//})=\frac{W_S}{W_{//}}$$

L'accélération de l'algorithme $A_{//}$ est définie comme le rapport du temps de A_S sur le temps $A_{//}$:

$$a(A_{//}) = \frac{T_S}{T_{//}}$$

L'efficacité de tout algorithme parallèle $A_{//}$ est bornée par : $0 \leq e(A_{//}) \leq 1$

Si on considère qu'un algorithme est constitué d'une partie purement séquentielle $T_{séq}$ et d'une partie parallèle T_{par} (loi de Amdahl), on peut écrire :

$$T_S=T_{séq}+T_{par} \text{ et } T_{séq}+\frac{T_{par}}{H_{//}} \leq T_{//}$$

On a donc les bornes suivantes sur l'accélération et l'efficacité :

$$a \leq \frac{T_{séq}+T_{par}}{T_{séq}+\frac{T_{par}}{H_{//}}} \text{ et } e \leq \frac{T_{séq}+T_{par}}{H_{//}*T_{séq}+T_{par}} \leq 1$$

On peut généraliser la loi de Amdahl : pour tout algorithme, supposons que l'on puisse identifier la partie purement séquentielle, la partie sur 2 processeurs, la partie sur 3 processeurs, etc. Le temps d'exécution séquentielle est $T_S = \sum_{i=1}^{mp} T_i$, où mp est le parallélisme maximum de l'algorithme. Le meilleur temps possible d'exécution parallèle est $T_{opt} = \sum_{i=1}^{mp} \frac{T_i}{i}$. Cette expression montre clairement que la qualité d'une implémentation peut dépendre de la divisibilité des tâches exécutables en parallèle à une étape de l'algorithme par le nombre de processeurs.

C. Contrôle du parallélisme

1. Ferme de processeur

a) Principe

La technique de contrôle du parallélisme appelée ferme de processeurs consiste à considérer qu'il existe un processeur maître qui va distribuer des données à traiter à ses différents processeurs esclaves qui lui communiquent en retour les résultats. Lorsqu'un esclave a terminé, il est disponible pour une nouvelle tâche. Le procédé est répété jusqu'à

épuisement du travail. Cette méthode correspond au cas où les processus (esclaves) n'ont pas besoin de communiquer entre eux. Cette technique est assez simple à mettre en oeuvre.

b) Technique d'implémentation

Vu la limite du nombre de liens sur un T800, un arbre ternaire (cf. figure 6.1) semble être la topologie la plus adaptée, car il utilise le plus possible de canaux de communication en parallèle, pour la ferme de processeurs.

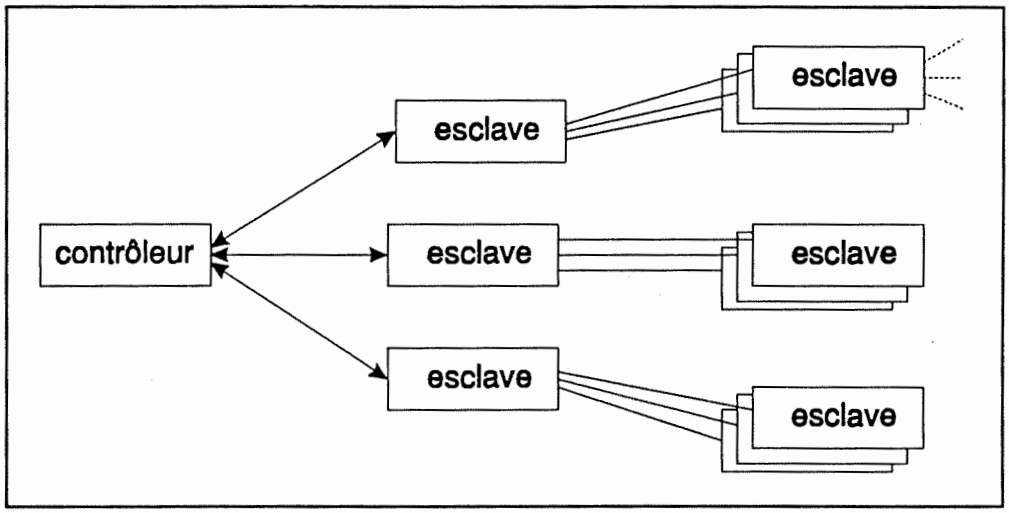


Figure 6.1. Un 3-arbre de processeurs.

Le principal défaut de cette technique est dû à la limitation du nombre de liens qui fait que quand on augmente le nombre d'esclaves, on arrive à un moment où les liens proches du contrôleur sont saturés par le passage des données entre le contrôleur et les esclaves.

Comme exemple concret d'utilisation d'une ferme de processeurs, on peut citer les astronomes de Southampton qui utilisent des transputers pour effectuer une simulation de Monte Carlo qui concerne des pluies de rayons gamma sur leur télescope. Ils utilisent le fortran EGS4 et obtiennent une accélération de 32 sur un ordinateur Meiko doté de 16 transputers comparé au même programme tournant sur un micro-vax II. Pour d'autres programmes qui utilisent intensivement les disques et les bandes magnétiques, le gain est moindre.

c) Exemple

Soit y, le vecteur résultat de la multiplication d'une matrice A nxn par le vecteur x.

$$\begin{pmatrix} a_{11} & a12 & \dots & a1n \\ a_{21} & a_{22} & \dots & \dots \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}$$

$$A \quad * \quad x \quad = \quad y$$

Chaque composante de y vaut : $y_i = \sum_{j=1}^{n} a_{ij} * x_j$ où $1 \leq i \leq n$.

Nous pouvons en observant que le résultat se décompose en n valeurs exprimer l'algorithme sous forme d'un ferme de processeurs où chaque esclave aurait comme tâche de calculer une composante y_i.

La tâche du maître consiste à envoyer à chaque processeur esclave une ligne de la matrice A (admettons que chaque esclave ait une copie du vecteur x) et à attendre le résultat.

2. Parallélisme géométrique

a) Principe

Le parallélisme géométrique utilise la structure sous-jacente des données d'un problème. Par exemple, de nombreuses méthodes, en particulier les très populaires méthodes de grilles sont aisément parallélisables, en confiant à un processus la charge de calcul d'une région à calculer. Dans ce cas, la subdivision du domaine en domaines plus

petits mais identiques entraîne que tous les processeurs ont la même chose à faire. On qualifie souvent ce parallélisme de parallélisme sur les données (*data parallelism*), ce qui correspond à un contrôle de type SIMD.

Cette constatation doit néanmoins être nuancée car elle ne s'applique que dans les cas où les interactions entre les sous-domaines sont locales. Si les communications sont plus globales (cas de l'algèbre linéaire), une augmentation du nombre des processus peut amener une chute de l'efficacité. Ce problème rejoint en fait celui plus général de l'équilibre qui existe entre la complexité des communications et celle des calculs.

b) Technique d'implémentation

La structure de données à traiter est répartie sur les processeurs selon un découpage régulier en sous-domaines. Les processeurs sont connectés en grille ou en tore et on attribue à chacun un sous-domaine. La programmation est simplifiée car chaque processeur exécute le même programme. Le transputer n'ayant que 4 liens, le découpage de l'espace d'origine ne peut se faire que selon, au plus, deux dimensions.

Le problème se complique si l'algorithme à répartir nécessite des communications à longue distance : il faut faire du routage. Ce cas se présente lorsqu'on veut programmer des algorithmes de l'algèbre linéaire : on a souvent besoin de communications point-à-point globales ou personnalisées (c'est-à-dire d'un sous-ensemble de tâches vers un autre), qu'il est difficile d'implémenter efficacement.

c) Exemple

(1) Description

Reprenons l'exemple du produit matrice-vecteur :

$$\begin{bmatrix} a_{00} & a01 & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ a_{(n-1)0} & a_{(n-1)1} & \cdots & a_{(n-1)(n-1)} \end{bmatrix} * \begin{bmatrix} x_0 \\ x_1 \\ \cdots \\ x_{(n-1)} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \cdots \\ y_{(n-1)} \end{bmatrix}$$

$$A \qquad * \quad x \quad = \quad y$$

La solution la plus simple pour implémenter ce produit sur une machine distribuée est de configurer la machine comme un anneau orienté (cf. figure 6.5) et de distribuer de manière équitable les lignes de la matrice A aux p processeurs. Chaque processeur reçoit les composants correspondants aux lignes de A et est responsable du calcul des mêmes composants du produit y.

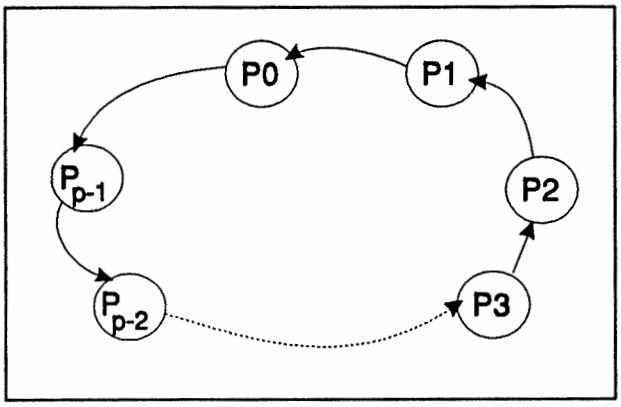


Figure 6.5. Anneau orienté de processeurs.

Cette solution requiert des communications inter-processeurs parce que chaque processeur a besoin de tous les composants de x pour calculer chaque composant de y.

$$y_i = \sum_{j=0}^{n-1} a_{ij} * x_j$$

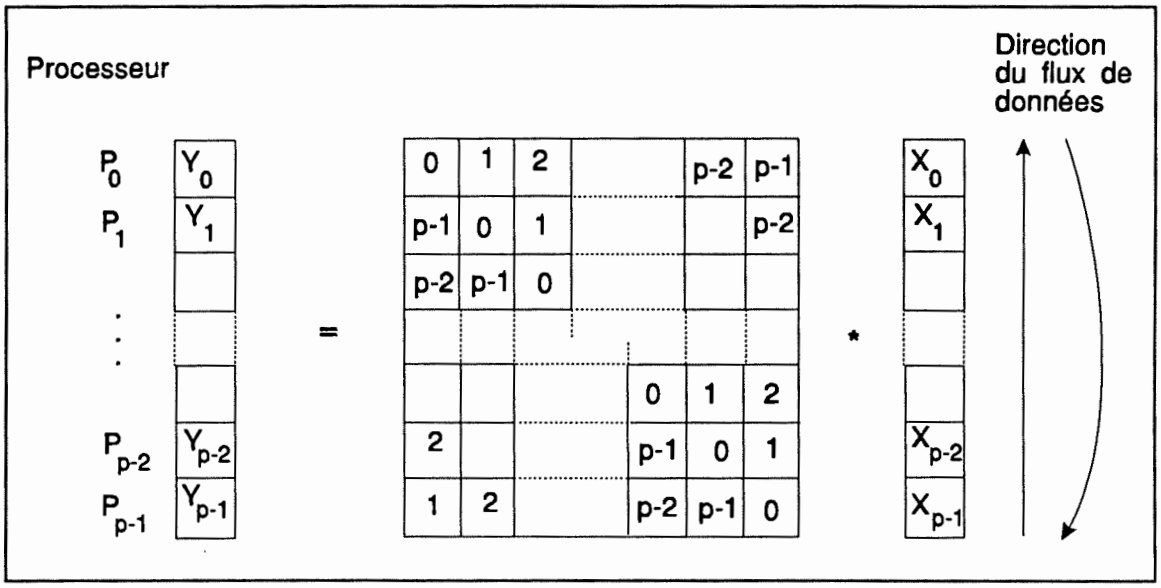


Figure 6.6. Produit matrice-vecteur.

Le processeur P_q, $0 \leq q \leq p$, contient le bloc de $r = \frac{n}{p}$ lignes consécutives d'indices q*r, q*r+1, ..., q*r+(r-1), P_q contient aussi les vecteurs X_q et Y_q des composantes correspondantes de x et y. Au pas s=0, P_q initialise le calcul de Y_q en utilisant le sous-bloc diagonal de A marqué avec des 0 dans la figure 6.6. A chaque pas s, où $0<s<p$, P_q reçoit un vecteur $X_{(s+q) \bmod p}$ de r composants de x de $P_{(q+1) \bmod p}$ et met à jour Y_q en utilisant le bloc marqué avec le pas correspondant au temps dans la figure. Après p-1 pas, le processeur P_q contient le sous vecteur résultat Y_q.

(2) Programme en pseudo-langage

Chaque processeur exécutera la tâche qui suit :

```
Tâche produit_mat_vec;

A : tableau [0..r-1,0..n-1];
x,y : tableau [0..r-1];
buf:tableau [0..r-1];

q: = proc_num();
pred = (q + 1) mod p;
succ = (q-1) mod p;
```

```
pour etape: =0 jusqu'à p-1 faire
        {
         en parallèle /* je vais lancer trois processus en parallèle */
                {
                envoyer(succ,x); /* j'envoie x au processeur suivant */
                recevoir(pred,buf); /* je reçois buf du processeur précédant */
                {calcul};
                };
        wait(recevoir);wait(envoyer);wait(calcul); /* j'attends la fin des trois processus */
        x: =buf;
        };
fin tâche
```

où {calcul} équivaut à :

```
pour i: =0 jusqu'à r-1 faire
        pour j: =0 jusqu'à r-1 faire
                {
                ind=((q+etape) mod p)*r+j;
                y[i]+ =A[i,ind]*x[j];
                };
```

(3) Programme en OUF

```
#define C3L
#define r 10
#define n 10
#define mod %
#include <ouf.H>
      ThreadDes processus[3]; /* Déclaration trois processus légers */
      ForkDes fourche; /* Déclaration d'une fourche */
      int A[r,n];
      int x[r];
      int y[r];
      int buf[r];
      int etape;

      void emettre() /* Déclaration d'une fonction qui deviendra un processus */
      {
      int temp;
      for (temp=0;temp<r;temp+ +) OutWord(x[temp],OutPort[0]);
       }

      void recevoir() /* Déclaration d'une fonction qui deviendra un processus */
      {
      int temp;
      for (temp=0;temp<r;temp+ +) InWord(&buf[temp],InPort[1]);
       }
```

```
void calcul() /* Déclaration d'une fonction qui deviendra un processus */
{
int i,j,ind;
for (i=0;i<r;i++)
  for (j=0;j<r;j++)
        {
        ind=((q+etape) mod p)*r+j;
        y[i]+=A[i,ind]*x[j];
        };
}

BEGIN_TASK("Pr. Mat_vec") /* Début de la tâche et déclaration du nom de la tâche */
#include "data.h" /* Données correspondant au numéro du processeur */
int q,pred,succ,temp;
ThreadsSpecif(1024,Low); /* les processus légers auront un espace de travail de 1024 bytes */
DefFork(fourche,3); /* La fourche comportera trois processus */
DefThread(processus[0],emettre); /* Définition d'un premier processus */
DefThread(processus[1],recevoir);
DefThread(processus[2],calcul);

for (etape=0;etape<p;etape++)
      {
      ForkThread(fourche,(processus[0])); /* lancement d'un premier processus */
      ForkThread(fourche,(processus[1]));
      ForkThread(fourche,(processus[2]));
      JoinThread(fourche);
      for (temp=0;temp<r;temp++) x[temp]=buf[temp];
      };

ENDTASK
```

(4) Configuration du réseau en Tn_Rita

En plus des processeurs que l'on retrouve dans la figure 6.5, j'ai rajouté une tâche se trouvant sur le transputer racine et qui peut faire des entrées/sorties avec l'extérieur, cette tâche est connectée de manière bi-directionnelle avec l'extérieur et avec la premier tâche de l'anneau de processeurs..

```
#include "makecfg.c" /* il s'agit d'inclure Tn_Rita */

#define p 5 /* Il y a p processeurs dans l'anneau */

main ()
      {
      Proc P[p]; /* Déclaration de np processeurs */
      Task io,T[p]; /* Déclaration de la tâche io et de p autres */
      int i; /* variable de travail */

      init_cfg(); /* initialisation de Tn_Rita */

      for (i=0 ; i<np ; i++) P[i]=DefProc(i); /*instanciation des processeurs */
      io=DefTaskIO("io",1,1);          /* instanciation de la tâche io, 1 canal en */
                                       /* et 1 en sortie. */
      BindTask(io,999); /* le numéro de la tâche io est 999 */

      for (i=0 ; i<=nbtach-1 ; i++) /* instanciation des tâches "matveci" */
            {
            Itoa(i,tempstr,10);
            T[i]=DefTask(strncat("matvec",tempstr,4),P[i],2,2);
            };


      DefBilink(io,2,T[0],2);    /* Déclaration d'un lien bi-directionnel */
                                       /* entre io et T[0] */

      for (i=0 ; i<nbtach ; i++) /* Déclarations de l'anneau uni-directionnel */
            Deflink(T[(i+1) mod p),0,T[i],1);
      imp_cfg(); /* Appel à Tn_Rita */
      }
```

(5) Complexité de l'algorithme

Le temps de {calcul} est $Tcalc=r^2\tau_\alpha$ où τ_α est le temps mis par les opérations arithmétiques ind=((q+etape) mod p)*r+j et y[i]+=A[i,ind]*x[j].

Le temps des communications (envoyer&recevoir une chaîne de taille r) est $T_{comm}=\beta+r\tau_c$ où β est le startup et τ_c le temps mis par l'envoi d'un mot.

Le temps d'exécution de l'algorithme avec recouvrement des temps de calcul et de communication (ce qui est réaliste pour des transputers) est :

$$T_p = p * \max\left(\frac{n^2}{p^2} * \tau_\alpha \, , \, \beta + \frac{n}{p} * \tau_c\right)$$

Le temps de l'algorithme séquentiel (pour i=1 à n faire $y_i = \sum_{j=1}^{n} a_{ij} * x_j$;) est $T_s = n^2 * \tau_{\alpha 2}$

Rappelons que l'accélération vaut $a = \frac{T_s}{T_{//}}$ et l'efficacité $e = \frac{T_s}{p * T_{//}}$

Pour calculer ce que ça donnera effectivement, nous pourrions prendre comme hypothèse que les lignes de la matrice comporte assez d'éléments pour que le temps de calcul soit prédominant sur le temps de communication, que nous possédons 128 processeurs, que $\tau_\alpha = 2\tau_{\alpha 2}$ (du fait du calcul de l'index), nous aurions alors une accélération de 64. Notons que si les éléments de la matrice et du vecteur étaient des nombres en virgule flottante, l'importance du temps de calcul de l'index diminuerait et l'accélération serait encore plus grande ($64 \leq a \leq 128$).

Remarquons que si on a p mémoires de taille M, la taille maximale du problème est $n^2_{max} < pM$.

3. Parallélisme algorithmique

Le parallélisme algorithmique est une généralisation du pipeline : l'algorithme est décomposé en étapes qui peuvent être exécutées en parallèle à condition qu'il soit possible d'acheminer simultanément des données à différentes étapes de traitement. Cette technique de contrôle est aussi appelée flot de données (*data flow*).

Par exemple, une chaîne de compilation (préprocesseur, analyseur syntaxique et sémantique, générateur de code, éditeur de liens) peut être distribuée sur plusieurs processeurs en chaîne. Sur cet exemple, il est difficile d'obtenir une efficacité meilleur que 30% ([Capon 90]). En effet, pour qu'un pipeline soit efficace, il faut que les différentes étapes aient le même temps d'exécution. Si une étape est trop longue, on devra la fractionner, ce qui peut poser un problème tout à fait artificiel et difficile.

Cette technique de parallélisation est de loin la plus ardue à exploiter efficacement, mais permet d'obtenir d'excellents résultats. Pour améliorer l'efficacité et l'accélération amenée par ce type de contrôle, il faut utiliser des techniques d'ordonnancement. L'exposé de ces techniques ne fait pas partie du présent travail.

VIII. Conclusion

Nous avons parcouru les différents types de machines parallèles. Ensuite nous avons vu les machines de type MIMD et plus particulièrement les ordinateurs parallèles à mémoire distribuée.

Nous avons étudié de manière pratique un environnement de programmation d'un réseau de transputers, le Méganode. Nous avons vu comment améliorer cet environnement de programmation grâce à une série d'utilitaires et nous avons regardé de plus près l'implémentation de l'un d'entre eux : le debugger.

Nous avons ensuite regardé les trois techniques les plus courantes de contrôle du parallélisme sur réseau de transputers.

L'étude des différents chapitres précédents et plus spécialement du dernier nous laisse entrevoir la difficulté de programmation (du moins de manière efficace) de telles machines. Cela explique pourquoi les sociétés hésitent à investir dans le domaine du parallélisme. De plus le fait de perdre tous les programmes déjà écrits les y fait généralement renoncer.

Le moyen de promouvoir le parallélisme serait de développer de bons outils de placement automatique des tâches et de parallélisation automatique. Remarquons que ces outils ne peuvent être facilement implémentés qu'à l'aide de ceux décrits précédemment. Nous pouvons donc conclure qu'après une première phase de réalisation d'outils de base (bibliothèque C, langage de description de réseau, debugger, ...), une deuxième phase logique pourrait être l'apparition de logiciels de parallélisation automatique.

Suite à cela, nous pouvons entrevoir une troisième phase où la généralisation de machines parallèles sera telle que les informaticiens se mettront à travailler dessus et que les théoriciens de l'informatique auront paufiné les techniques de programmation parallèle.

IX. Bibliographie

[Adamo 90], Jean-Marc Adamo et Christophe Bonello, "TéNOR : a symbolic configurer for SuperNode architecture", Laboratoire LIP-IMAG, email adamo@ensl.ens-lyon.fr, 90.

[Banatre 90], J.P. Banatre,"La programmation parallèle", mai 1990.

[Behr 89], P.M. Behr, W.K. Giloi et W.Schröder, "Synchronous versus asynchronus communication in high performance multicomputer systems" in "Aspect of computation on asynchronous parallel processors" édité par M. Wright chez North-Holland, 89.

[Bell 89], Gordon Bell, "The future of high performance computers in science and engineering", communications of the ACM, septembre 89.

[Carriero 89], Nicholas Cariero, David Gelernter, "How to write parallel programs : A guide to the perplexed", ACM computing surveys, vol 21, n°3, septembre 89.

[Capon 90], P.C. Capon, "Experiments in algorithmic parallelism", 11 th OUG technical meeting, Edinburgh, IOS, 1990.

[Dewar 90], Robert B.K. Dewar, Matthew Smosna, "Microprocessors, a programmers view", Mac Graw Hill, 1990.

[Dinning 89], Anne Dinning, "A survey of synchronization methods for parallel computers", IEEE Transactions on computer, juillet 89.

[Duncan 90], Ralph Duncan, "A survey of parallel computer architectures", IEEE Computer, février 90.

[Eager 89], Derek L. Eager, John Zahorjan, Edward D. Lazowska, "Speed-up versus efficiency in parallel systems", IEEE transactions on computers, vol 38, n°3, mars 89.

[Gehani 88], Narain Gehani, Andrew D. McGettrick, "Concurrent Programming", Addison-Wesley, 1988.

[Gomm 91], Dominik Gomm, Michael Heckner, Klaus-Jörn Lange, Gerhard Riedle, "Some aspects of distributed memory computing", EDMCC2, avril 91.

[Hey 89], Anthony J.G. Hey, "Experiments in MIMD parallelism", invited paper at the 1989 PARLE conference, Eindhoven.

[Hirsh 90], Ernest Hirsh, "Les transputers, application à la programmation concurrente", Editions Eyrolles, 90.

[Hoare 85], C.A.R. Hoare, "Communicating sequential processes", Prentice-Hall, 1985.

[Hwang 84], Kai Hwang, Fayé A. Briggs, "Computer Architecture and parallel processing", McGraw-Hill, 84.

[Inmos 88], INMOS, "Transputer reference manual", Prentice Hall International (UK), 1988.

[Karp 90], Alan H. Karp, Horace P. Flatt, "Mesuring parallel processor performance", communications of the ACM, mai 90.

[Kordon 90], Fabrice Kordon, "Les machines RISC".

[Kung 89], H.T. Kung, "Computational models for parallel computers", édité par R.J. Elliot et C.A.R. Hoare, Prentice-Hall, 89.

[Litaize 90], D. Litaize, "Architectures multiprocesseurs à mémoire commune", deuxième symposium architectures nouvelles de machines, PRC ANM, CNRS, MRT, Toulouse, septembre 90.

[Perrot 87], R.H. Perrot, "Parallel programming", International Computer Science Serie, Addison-Wesley, 87, réimprimé en 88.

[Pountain 88], Dick Pountain & David May, "A tutorial introduction to Occam programming", Inmos, 1988.

[Pountain 89], Dick Pountain, "Configuring parallel programs", Byte, décembre 89 et Janvier 90.

[Pountain 90], Dick Pountain, "Virtual Channels : The Next Generation of Transputers", Byte, avril 90.

[Pritchard 89], David J. Pritchard, "Mathematicals models of distributed computation" in "Aspect of computation on asynchronous parallel processors" édité par M. Wright chez North-Holland, 89.

[Raynal 90], Patrick Raynal, "Placement de tâches sur une architecture multi-processeurs", DEA en informatique, INPG, 90.

[Robert 90], Ken Grigg, Serge Miguet, Yves Robert, "Symmetric matrix vector product on a ring of processors", Rapport 90-06 du LIP (Laboratoire de l'informatique du parallélisme), ENSL (Ecole Normale Supérieure de Lyon).

[Snyder 90], Lawrence Snyder,"New programming abstractions for distributed memory parallel machines, University of Washington, Seattle, 90.

[Tanenbaum 89], Henri E. Bal, Jennifer G. Steiner, Andrew S. Tanenbaum, "Progamming languages for distributed computing systems", ACM computing surveys, vol. 21, n°3, Septembre 89.

[Touzene 90], Abderezak Touzene, Brigitte Plateau, "Mesure de performance des communications du Méganode à 128 transputers", LGI (Laboratoire de Génie Informatique)-IMAG, 90.

[Trystram 90], Denis Trystram, François Vincent, "Programmation avancée du Transputer : architecture et mécanisme", email trystram@afp.imag.fr.

[Waille 90], Ph. Waille, "Introduction à l'architecture des machines supernode", Rapport Technique, RT56, de l'IMAG, février 90.

[Zampuniéris 90], Denis Zampuniéris, "Analyse sémantique des communications entre processus de programmes parallèles de type CSP", rapport FUNDP & Ecole Polytechnique de Paris, 1990.

[Zitterbart], Martina Zitterbart, "Monitoring and debugging tansputer-networks with Netmon-II", Institute of Telematics, University Karlsruhe, Kaiserstr. 12, 7500 Karlsruhe, FRG.

X. Index

A
accélération, 79
All To All, 23
All To One, 23
alternative, 43;56;67
anneau, 19
arbre, 20
arbre de recouvrement, 18
asynchrone, 36;37

B
bande passante, 14;16
bibliothèque, 65
boîtes aux lettres, 36
bus, 14

C
C004, 52
C104, 49
canal, 42;66
CCC, 17
CISC, 9
codes de Gray, 17
commande gardée, 38
communication, 26;28;56
communications, 23
complètement connecté, 49
configuration, 56;62
connexions statiques, 16
contention, 14
contrôle du parallélisme, 80
crossbar switches, 21;47
Cube Connecté Cycle, 17

D
daisy chaining, 14
data-flow, 11
debugger, 67
debugger post-mortem, 69
diamètre, 16
diffusion, 25

E
efficacité, 79
environnement, 60
Esprit 1085, 52

E
étage, 7

E
exclusion mutuelle, 26;66

F
FCFS, 15
ferme de processeurs, 80
flot de données, 11
FPU, 46

FTS, 14

G
garde, 38
grille, 20

H
hyper-cube, 17;19

L
lien physique, 48
liens, 16
loi de Amdahl, 79
LRU, 15

M
maître-esclaves, 20
Méganode, 61
mémoire distribuée, 16
mémoire locale, 13
mémoire partagée, 13
messages, 35
MIMD, 10;13
moniteurs, 32

N
nomination directe, 35
nomination globale, 36

O
OCCAM, 42
Occam, 45
One To All, 23
One To One, 23
ordonnanceur, 47
Ouf, 65

P
parallélisme, 1;4
parallélisme algorithmique, 89
parallélisme géométrique, 82
partage du temps, 27
pipeline, 7;10;11;24;25;89
ports, 36
priorité, 14
processus léger, 65
programmation, 77
programme concurrent, 27
programme séquentiel, 27
protocole, 48
pseudo-parallélisme, 4;42

R
rendez-vous, 38
réseau complètement connecté, 49
réseau d'interconnexion, 13
réseaux fixes, 16
réseaux systoliques, 12
RISC, 9;46

S
sémaphore, 39;67
sémaphores, 28;67
sémaphores binaires, 29
sémaphores faibles, 29
sémaphores forts, 30
sémaphores généraux, 29
SIMD, 5;11
SPMD, 5
sun4, 55
surface, 78
synchrone, 36;37
synchronisation, 26;28
synchronisme fort, 37
systolique, 12

T
T800, 44
T9000, 49
tâche, 55;65
TDM, 14
temps, 78
threads, 55
Tn_Rita, 62
tore, 20
transputer, 45

U
UC, 6
unité de contrôle, 6
Unité flottante, 46

V
variables partagées, 27